(9)

SYNCHRONIZATION OF FINITE STATE SHARED RESOURCES

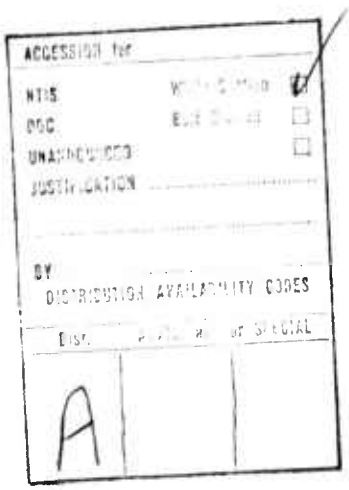
Edward A Schneider

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

See Form 1473

SYNCHRONIZATION OF FINITE STATE SHARED RESOURCES



Edward A Schneider

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
March, 1976

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074, and in part by the National Science Foundation under contract GJ 32259. This document has been approved for public release and sale; its distribution is unlimited.

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

ABSTRACT

The problem of synchronizing a set of operations defined on a shared resource is studied. It is assumed that the decision as to which operations may be executed at some given time is dependent only on the sequence in which the operations have already executed. Equivalence classes of these sequences, called states, can then be used to define synchronization. A restriction is made such that only those resources for which the synchronization can be expressed using a finite number of states will be studied. The states along with a successor function, which is defined for a state-operation pair if the operation may start execution when the resource is in that state, form what are called synchronization relationships.

A distinction is made between resources on which only one process may execute an operation at a time, called serial resources, and resources on which several processes may execute operations in parallel, called concurrent resources. To handle concurrent resources, the states must be modified so that they correspond to equivalence classes of sequences of perilogues instead of operations. A perilogue is either the start or the finish of the execution of some operation.

Several variations of regular expressions are presented with which the synchronization for a shared resource might be expressed. Also, a method which can be used to implement the synchronization relationships is given. This implementation has a high overhead so several possible simplifications are shown. Each variation of regular expressions and each simplification of the implementation is shown to

correspond to some restricted class of the synchronization relationships. The set of synchronization problems which can be solved using one implementation or notation which can't be solved using some other implementation or notation can be found by comparing the corresponding classes.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and thanks to my advisor and thesis committee chairman Professor A. N. Habermann for introducing path expressions and for the many hours that he has spent discussing this work with me. I also wish to thank the rest of my thesis committee, Professors A. K. Jones, N. Suzuki, and P. Andrews for their constructive suggestions.

Finally, I want to thank my wife Ann and daughter Peggy for their patience and understanding.

CONTENTS

CHAPTER I

INTRODUCTION

In programming systems, it is usually necessary to enforce rules regulating the behavior of the system. Such a set of rules is known as a protection policy and the enforcement mechanisms are known as a protection system implementing that policy. Scope rules in programming languages and the restriction of allowing only authorized users to access files are examples of protection. Another example occurs when one of several cooperating processes must temporarily halt execution pending the completion of some set of actions by the others. Such protection is referred to as synchronization. A set of rules governing when a process must halt and when it can continue execution is referred to as a synchronization problem.

An important use of synchronization is to control the access to resources by cooperating sequential processes. A resource is any physical device or segment of memory which can be referenced by the programming system. Some examples are a data structure in a process' virtual memory, a file on a permanent storage device, and an I/O device. Each resource has associated with it a set of operations which are used to extract information from it, to alter information in it, or to add information to it.

In order for the processes to cooperate, it is necessary to allow them to share resources. For instance, a message buffer must be accessible by the processes sending messages and by the processes receiving them. To insure that the value of a shared resource is always well defined when a process invokes one of the operations

defined on it, usually the resource may only be operated on by one process at a time and the operations must execute only in certain sequences. If some process tries to execute an operation while some other process is executing on the resource or an incorrect sequence of operations would result, then the process must halt until this condition is no longer true. Such synchronization on shared resources is the topic which will be explored.

BACKGROUND

The problem of synchronizing processes without using busy waiting was first solved by Dijkstra with P and V synchronization primitives [D68]. Since then, several other synchronization primitive sets have been proposed to solve problems which couldn't be solved easily with any of the existing primitives. These include allowing P or V to be executed simultaneously on several semaphores (P-V multiple) [P71] (introduced to solve the cigarette smokers problem), allowing a semaphore to be incremented or decremented by a value greater than one (P-V chunk) [VL72] (introduced to solve the bounded reader-writer problem), and separating the testing and the decrementing of a semaphore into two operations (UP-DOWN) [W72] (introduced to solve the general reader-writer problem).

In addition, several methods have been proposed to allow synchronization to be expressed in a more "structured" manner. These methods are meant to be used in a high level language to control access to shared resources. An analogy can be made between the relation of these high level methods to the primitives and the relation of high level programming language control structures to assembly language. The high

level synchronization methods provide a structured means for expressing frequently occurring synchronization, thus helping to improve understandability and reliability, just as IF - THEN - ELSE and WHILE - DO statements provide a convenient way to express frequently used control paths. A compiler can then be used to implement the synchronization in terms of primitives just as the control structures are implemented in terms of test and jump instructions.

The first of these methods is "regions" proposed by Brinch Hansen [BH72]. A region is a statement type that is associated with some shared resource. For each of these resources, only one process at a time can execute an associated region. Furthermore, shared resources can only be accessed in these regions. Thus, regions provide structured critical sections and allow a compiler to enforce mutually exclusive access of shared resources. Sequencing is still handled using primitives.

It has long been recognized that operations composed from simpler ones should only be executed in a restricted manner Thus, procedures have only one entry point and a jump may not be made into the middle of one from outside of it. Likewise, primitive data types such as integers, reals, booleans, and characters may only be operated on by certain operations. For instance, booleans may not be added. This idea should be extended to more complex resources. They should only be accessible through a few operations which completely define the behavior of the resource.

Allowing a resource to be accessed only by some fixed set of operations has several advantages. The first is that at the point it is used, all that needs to be known about the resource is the effect of each operation which can be used. How it is implemented is unimportant. For instance, a stack is defined by the effects of the

operations push an item, pop an item, and test for emptiness on the values of the other operations. Whether it is implemented using an array, a linked list, or by some other means is unimportant. Users perceive only the three operations.

Next, if the resource may only be accessed through several operations rather than in an arbitrary manner, it is more likely that the value of the resource will always be meaningful. Finally, a verification that the resource always is accessed correctly can be confined to several operations and can ignore the rest of the code of the programs which use the resource.

A convenient means to insure that a resource can only be accessed through a fixed set of operations is to include the implementation details and the operations in a module. The only names defined in the module which can be used outside of it are those of the operations. Concentrating the implementation and access details of a resource in a module also has the advantage that if the implementation is changed in some way, all of the places in the access algorithms that need modification are localized and are therefore easily found. There is no need to search through all of the programs that use the resource to make these changes. Flon [F75] discusses such modules in more detail and gives some examples.

The module is also the best place to define in which sequences the operations of a resource may execute. The synchronization can then be considered as part of each operation and the operations can be used without concern for synchronization. Also, as with the implementation, it is easier to make modifications and to verify correctness. The last two synchronization methods to be described are meant to be used in just this way.

Hoare [Ho74] proposed the concept of a "monitor" for synchronization. A monitor is a collection of data, procedures which operate on this data, and initializations. The data may only be accessed by these procedures and only one process at a time can do so. Thus, a monitor may be thought of as providing a critical section around the data and procedures of a shared resource. As with regions, sequencing still must be expressed using primitives.

Finally, Campbell and Habermann [CHa74] have proposed path expressions as a means of synchronizing a set of procedures operating on a collection of data. A path expression consists of an expression R which contains each operation name once and which is enclosed in a PATH - END pair. R may be a single operation name, it may take the form R'+R" meaning that either some sequence of operations expressed by R' or one expressed by R" may occur, or it may take the form R';R" meaning that some sequence of operations expressed by R' should be followed by one expressed by R" where R' and R" are of the same form as R. The path expression, once completed, may then be repeated. Thus,

PATH f ; (g+h) END

means that f should be executed first, then g or h, and then this sequence starts over again.

MOTIVATION

To be able to decide how useful a given method is for some application, it must be known which of the desired synchronization problems can be solved using that

method. Therefore, when a synchronization method is proposed, the class of problems which can be solved using it should also be given. Lipton [L73] has compared the various primitives and for each one has characterized some of the problems which can't be solved using that primitive system. The complete set of problems which each synchronization method can or can't solve hasn't been shown, though.

A strong meaning of "solve" must be used here since any synchronization primitive may be implemented using critical sections and letting a process block itself or wake up a blocked process. The following is an example of how this can be done:

```
CRITICAL SECTION BEGIN
test each blocked process
IF process-j may now continue THEN WAKEUP(process-j) FI;
IF this process can't continue THEN indicate it is blocked
                                    CRITICAL SECTION END
                                    BLOCK
                                ELSE CRITICAL SECTION END
FI
```

The BLOCK occurs outside of the critical section in order to allow other processes to enter to execute WAKEUP. One way to find each blocked process is by keeping a list of them. Then, to indicate that a process is blocked, it is put on this list. Notice that a process which wants to wake another might be stopped trying to enter the critical section if another process is already in it. Such a delay wouldn't occur if a primitive replaced the critical section. Therefore, by solve it will be meant that there aren't any extra places where a process may become blocked such as at the start of the critical section above.

In order to show that a group of processes cooperate correctly, it must be possible to understand how they are synchronized. Also, the consequences of any

modifications to the synchronization must be understandable. This helps insure that what is actually programmed is what was desired. It also makes it easier for someone else to make changes. As the difficulty in understanding increases, the possibility of an error occurring and the difficulty in detecting such errors also increase. Finally, certain deadlock possibilities should be detected. These possibilities include a process that waits on a semaphore which has an initial value of zero and which no process will ever increment. Another example is when a process uses a critical section nested in another and a second process uses the second critical section nested in the first.

The problem with synchronization primitives is that they, and therefore any changes, may be scattered throughout the code executed by the various processes. Furthermore, no structure is imposed on their use. Regions and monitors provide higher level structures for writing critical sections, but sequencing must still be performed using synchronization primitives. These primitives may be scattered throughout a monitor or region. Only with path expressions is the desired sequencing clear.

Another problem with regions and monitors is their strict enforcement of mutual exclusion. In the reader-writer problem where the read operation may be performed simultaneously by an arbitrary number of readers, this operation can't be part of any region or monitor. This means that the data structure on which the read operation is defined can't be part of any monitor since otherwise any operation which can execute on it must also be part of that monitor. The result is that operations "startread" and "endread" must be introduced just to provide synchronization. Path expressions solve this problem with the introduction of a {-} construction. This notation has the meaning

that an arbitrary number of processes may execute the operations within the brackets simultaneously. Thus,

PATH write+{read} END

means that either one process may write the data structure or several processes may simultaneously read it. The brackets, however, don't allow restricting the number of readers to some finite bound.

PROBLEM TO BE STUDIED

It is the purpose of this research to study synchronization in terms of the allowable sequences of operations on a shared resource. Thus, it will be assumed that each resource may only be accessed through a fixed set of operations. Since state machines have been widely used to study sequences of symbols [HU69], it will be convenient to use them to represent these sequences. Each operation defined on the resource will correspond to one or more state changes. In order to simplify the study somewhat, only that synchronization which can be described in terms of a finite number of states will actually be discussed.

The operations which can be used on a shared resource are executed by the various processes of the programming system. A process can be considered to be a sequence of calls on the operations of the shared resources possibly interspersed with calls on the operations of resources which can only be accessed by that process. There is also some control which regulates the sequence of operation calls.

A distinction will be made between those resources on which operations can be executed in parallel and those on which operations must be executed one at a time. In order to handle parallel execution, each operation must consist of two state changes, one at the beginning of the operation and one at the end.

Several subclasses of finite state machines will be introduced by restricting the admissible state changes. Since each synchronization problem is represented by a state machine, each of these subclasses limits the set of problems which can be expressed. Therefore, each restriction of the finite state machines also defines a class of synchronization. The task of showing which problems a synchronization system solves thus corresponds to presenting the appropriate restriction of the state machines.

The main criterion which is used to restrict the state changes is the manner in which the resulting synchronization class can be implemented. If for some implementation there is no corresponding class, then every class containing the set of synchronization problems which could be implemented with that implementation which contains this set must also contain some synchronization problems which can't be. Thus, a more complex implementation is needed for every such class. However, if this set contains all of the synchronization problems of interest, then the simpler implementation would have been sufficient.

Ease of implementation shouldn't be the only factor used in selecting the class of synchronization to provide in a language for parallel programming. In order to express synchronization outside of the class which is provided, a user must implement a larger class in terms of the existing class. The resulting implementation must be more

complex than if the larger class had been provided initially. Furthermore, the user has an extra opportunity for a programming error.

In order for the designer of a language for parallel programming to be able to use one of these classes, it must be possible to express the synchronization of that class in some notation. As explained above, path expressions provide a means for expressing synchronization which is easy to understand relative to the other methods. Unfortunately, they can only be used for a simple class of problems. Regular expressions of the operation names, a generalization of path expressions, can be used to describe any synchronization which can be expressed with a finite state machine. This suggests that some restriction to regular expressions would be suitable for each class. Therefore, several modifications to regular expressions will be introduced and compared with the classes.

Even though regular expressions are used in this research, there might exist other notational systems which are equally suitable. Regular expressions were selected because of their correspondence to finite automata and because they are easy to use and understand for simple synchronization. If some other notation is used, the class of synchronization which can be expressed with it should be shown.

OUTLINE OF THE THESIS

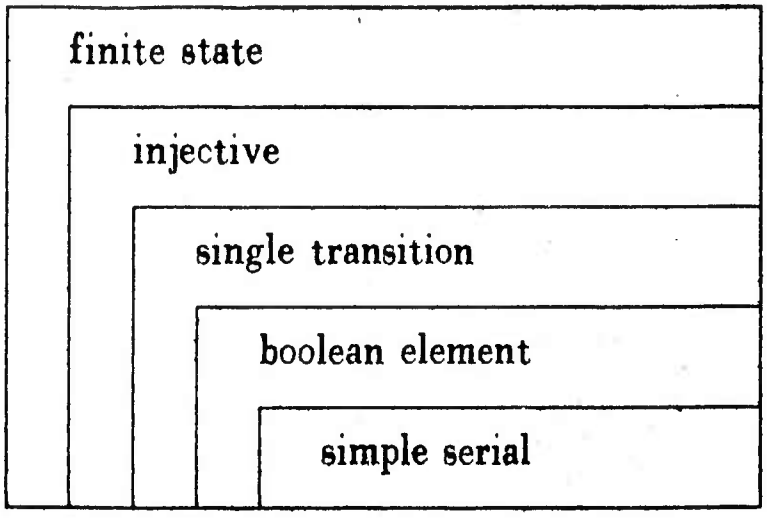
In chapter II, the finite state model for resources on which only one operation at a time may execute is developed. This includes some definitions and basic results as well as a discussion of how these resources may be implemented. Where relevant,

how a priority system among the various operations might be handled is also discussed. Finally, several restrictions to the model are presented. Chapter III continues the discussion of sequential resources with the Z theorem. A class of regular expressions is presented in which operation names may not occur more than once and in which there are restrictions as to where the R^* notation may occur. These expressions are shown to correspond to a class of finite state graphs with a simple implementation. This result is then extended to programming language control structures.

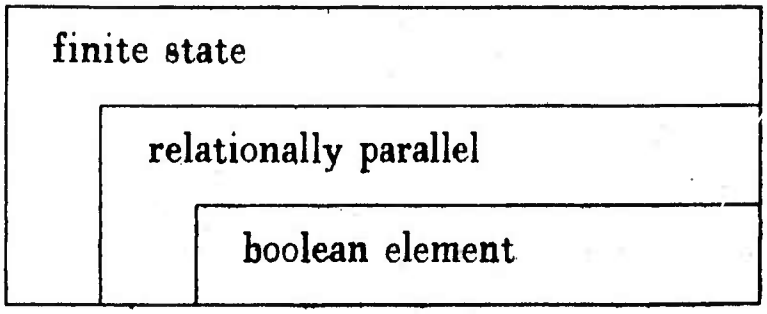
In chapter IV, the notion of a state is changed to be a multiset of objects which will be called elements. These elements are then used to define several more restrictions to the finite state model. In chapter V, resources on which several operations may execute in parallel are studied. The elements introduced in chapter IV are used to simplify the implementation of these parallel resources. Chapter VI summarizes what has been shown and points to areas where future research is needed.

The figure on the next page shows how the restrictions which are presented are related to each other.

SERIAL RESOURCES



CONCURRENT RESOURCES



CHAPTER II

SERIAL RESOURCES

The first type of synchronization to be studied deals with shared resources on which not more than one process may execute operations simultaneously.

Definition: *A serial resource* is a shared resource on which only one process at a time may execute.

It should be noticed that for a serial resource, no operation f for that resource may use any other such operation g. Otherwise, using f would require that g execute during the execution of f.

It may also be the case that the appropriate operations must execute in specific sequences.

Example 2.1: Consider a buffer used to pass messages between processes. The operations which are defined are "insert a message" and "remove a message". If more than one operation may execute at a time, messages could be inserted jumbled together or only part of a message might be removed. If each message must be received exactly once, then the execution of remove and insert must alternate. Otherwise, a message might be overwritten and lost or else it might be received twice.

It is the responsibility of whatever synchronization system is used to guarantee this serial execution and to insure that an incorrect sequence of operations doesn't occur.

For some serial resources, any sequence in which the operations execute is acceptable. If example 2.1 is modified so that a message may be received more than

once or not at all, then all that matters is that Insert and remove don't overlap in time. The order in which they execute is no longer important. Such synchronization is usually handled by writing the operations as critical sections.

One way to express the allowable sequences of operations is to write them out explicitly. Alternatively, relationships of the form "operation f may be executed on the shared resource if the order in which operations have been executed form sequence α" may be used. However, if there is no restriction on the number of times that an operation may execute, these sequences may be arbitrarily long. Therefore, an infinite number of these relationships would be necessary. Usually, though, part of the previous history is unimportant.

Example 2.2: Returning to the message buffer of example 2.1, the desired sequencing is that the execution of insert and remove alternate. Therefore, when something is removed from the buffer, it matters only that the most recent operation on the buffer was insert and when something is inserted, it matters only that the most recent operation was remove.

Definition: The *state* of a shared resource is that part of the succession of operations which have executed on the resource and that is necessary to determine which operations may execute in the future.

In what follows, the symbols p and q will usually be used to represent the state.

The relationships now take the form "operation f may be executed on the shared resource if its state is p with the result being state q".

Definition: The *synchronization relationships* for a shared resource consist of a list of the states and for each, a list of the operations which may execute when the resource is in that state and the state which results.

The resulting state q is created by adding f to the execution sequence represented by p. Of course, some of this history may no longer be important and will be omitted from state q.

The following notation will be useful when dealing with these relationships.

Definition: If p is a state and f an operation, then the successor function, $S(p,f)$, has the value q if operation f may execute when the resource state is p with the resulting state being q. If f can't execute when the state of the resource is p, then S(p,f) is undefined. If S(p,f) is defined, then (p,f) is an *arc* of the resource.

Example 2.3: For the message buffer, example 2.2 shows that there are two states lastinsert and lastremove with lastremove being the initial state, S(lastremove,insert) = lastinsert, and S(lastinsert,remove) = lastremove. S(lastremove,remove) and S(lastinsert,insert) are undefined. The arcs are (lastremove,insert) and (lastinsert,remove).

Thus, the successor function S is a partial function which is defined for those states and operations such that the operation may execute when the resource is in that state. Whenever S(p,f) is defined, it will be said that operation f may be *applied* at state p.

It will often be desirable to determine if a sequence of operations, rather than a single operation, may execute on a shared resource.

Definition: An *arc progression* from a state q_0 to a state q_n is a string of arcs $(q_0,f_1)...(q_{n-1},f_n)$ such that $(\forall i, 1 \le i \le n)\ S(q_{i-1},f_i) = q_i$.

Thus, an arc progression specifies a possible ordering for the execution of the operations f_i. Note that there is no restriction requiring that the arcs be distinct. It might be true that $q_{i-1} = q_{j-1}$ and $f_i = f_j$ for some i and j, $0<i<j\le n$. When this happens, it must also be true that $q_i = q_j$. A special case is when an arc progression is circular.

Definition: A *cycle* is a non-empty arc progression from a state q to q.

In example 2.3, the arc progression (lastremove,insert)(lastinsert,remove) is a cycle.

Critical sections are an even simpler case of cycles. Since any sequence of operations is acceptable, none of the previous history is important. Therefore, a single state is sufficient and each operation must start and end at this state. Each arc progression, including any of length one, is from this single state to itself and is a cycle.

FINITE STATE RESOURCES

As states have been described so far, it is impossible to deal with an infinite number of them. The successor function is defined by listing the value for each arc of the resource. If the number of states is infinite, then so is the number of arcs. Thus, a natural restriction will be to permit only a finite number of states for a shared resource.

Definition: A resource is ***finite state*** if the number of states, and therefore the domain of the successor function, is finite.

Unfortunately, there are serial resources with an infinite number of states. Consider a stack of unbounded size on which the operations PUSH and POP are defined. The desired synchronization is that only one process at a time can execute one of these operations and that at any given time PUSH must have been executed at least as many times as POP. The information represented by the state must be how many more times PUSH has executed than POP. Since this number may be arbitrarily large, there must be an infinite number of states.

Usually, such resources may be studied with a finite state system by putting a limit on the memory size used by such a resource or, if the resource isn't serial, then on the number of processes which can use the resource simultaneously. Thus, the size of the stack in the above example could be bounded. Such a restriction would occur in practice anyway. A mechanism will be developed in chapter IV which will enable the handling of some resources with an infinite number of states and an indication of how this can be done will be given in chapter VI. Other than in these places, however, such resources will be outside the range of the research reported here.

In order to help study finite state resources, the concept of a finite automaton [HU69, page 26] is needed. A finite automaton is a system (K,Σ,δ,q_0,F) where K is a nonempty, finite set of states, Σ is a finite input alphabet, δ is a mapping of (K,Σ) into K, $q_0 \in K$ is the initial state, and $F \subseteq K$ is the set of final states. The system is initially in state q_0 and as each successive character f_i of an input string is read, the automaton enters state $q_i = \delta(q_{i-1},f_i)$. If $q_n \in F$, then the string $f_1 \ldots f_n$ is accepted. Otherwise, it is rejected.

While the synchronization relationships for a finite state resource resemble a finite automaton, there are several differences. These differences are based on how each is used. A finite automaton is used to indicate whether or not a given string is correct. Thus it has final states. Also, regardless of what state the automaton is in, any input is possible and therefore a resulting state must be defined. However, if an input insures that the string will be rejected, it must be impossible to reach a final state from the resulting state.

Definition: A state $p \in K$ is *dead* if $(\forall x \in \Sigma^*)\ \delta(p,x)$ isn't a final state.

In the definition, Σ^* is the set of all strings of length 0 or more of symbols from Σ. The function δ is extended to Σ^* as follows. If x is the string of length 0, then $\delta(p,x) = p$. If $x = x's$ where $x' \in \Sigma^*$ and $s \in \Sigma$ then $\delta(p,x) = \delta(\delta(p,x'),s)$.

The easiest way to find the dead states is to first find the set L of states which aren't dead. Clearly, any state of F is in L. Therefore, L is initialized with F. Any state q such that $(\exists s)\ \delta(q,s) \in L$ is also in L. This procedure is then repeated until there are no more such states q. Any states which aren't in L at this point are dead.

The synchronization relationships, on the other hand, are used to guarantee that only correct strings are input. Any input which would insure that the string is incorrect is delayed until this condition no longer holds. Thus, not every input is possible from any given state and in such cases a resulting state is not defined. This means that dead states aren't needed. Finally, usually an infinite string will be input so the idea of a final state is meaningless.

Subject to these restrictions, the following result is presented.

Theorem 2.4: A serial resource R is finite state iff the synchronization relationships and some finite automaton (K,Σ,δ,q_0,F) represent the same acceptable sequences of symbols.

Proof: For each state p of R, let there be a state $p' \in F$ and for each operation f of R let there be a symbol $s \in \Sigma$. In addition to the states of F, let there be another state in K which is dead. Since the number of states and operations of R are finite, so are the number of states and the input alphabet of the finite automaton. Define $\delta(q,s)$ as follows. If S(p,f) is defined, then $\delta(p',s)$ is the state of F corresponding to S(p,f). Otherwise,

$\delta(p',s)$ is the dead state. The construction is completed by letting the initial state of K be the state corresponding to the initial state of R. A set of synchronization relationships corresponding to a finite automaton may be created by reversing this process.

This correspondence between the synchronization relationships for finite state serial resources and finite automata can be used to apply results from automata theory. Two elementary results are particularly important.

There is a class of expressions, known as regular expressions, which have been shown to represent the same class of strings from an alphabet as can be recognized by finite automata [HU69]. These expressions may be described recursively as follows. A single character from the alphabet is a regular expression. So are constructs of the form RR', R+R', and R^* where R and R' are also regular expressions. RR' means a string represented by R followed by a string represented by R'. R+R' means either a string represented by R or a string represented by R'. R^* represents the infinitely long expression ϵ+R+RR+RRR+... where ϵ is the empty string. The following result can now be given.

Corollary 2.5: A serial resource is finite state iff the permissible sequences of operations on it can be expressed using a regular expression.

For example, the synchronization for the message buffer of example 2.3 can be expressed with the regular expression (insert remove)*.

Using a regular expression rather than the synchronization relationships to specify synchronization has several advantages. First, the system designer no longer

needs to worry about states. Second and more important, it is easier to understand which are the allowable sequences of operations.

The unimportance of final states has an effect on the regular expressions which can be used to specify synchronization. The expressions $(f^*g)^*$ and $(f|g)^*$ both indicate an arbitrary interleaving of the execution of the operations f and g. The difference is that in the first expression, the state won't be final if the last operation to execute was an f. In the second, though, there is a single state which is also a final state. For use in synchronization, since final states are unimportant, these expressions are equivalent.

The successor function as described is determinstic. By this is meant that for each element of the domain either the result is unique or else it is undefined. If the successor function were nondeterministic, there would be more than one possible result for some argument. A state would be chosen at random for which there might be no processes waiting. However, processes could be waiting for another possible resulting state. These processes would then continue to wait even though it would be permissible to allow one to run.

It might be worth considering a nondeterministic successor function if some synchronization can be described with a finite number of states that would require an infinite number if the successor function is deterministic. The following result from automata theory shows that there are no such resources.

Corollary 2.6: If a serial resource is finite state, then the allowable sequences of operations on it can be expressed using a deterministic set of synchronization relationships.

If the successor function S is nondeterministic, then a set of synchronization relationships with a deterministic successor function S' which expresses the same allowable sequences of operations can be constructed as follows [HU69]. For each nonempty element of the power set of states $\{p_1,...,p_n\}$, create a new state q. Assume that for operation f $S(p_1,f) \cup ... \cup S(p_n,f) = T$ where T is a set of states. If T is nonempty, then there must be some new state q' which corresponds to T. In this case, let $S'(q,f) = q'$. If T is the empty set, then f can't be applied at q and S'(q,f) is undefined. The new initial state q_0 is the state which corresponds to $\{p_0\}$ where p_0 was the original initial state. The synchronization relationships can be simplified by removing every state to which there is no arc progression from q_0.

PERSISTENT SETS

For programs consisting of several parallel processes which may run for an indefinite period of time, such as an operating system, some of the operations defined on each resource must be able to be executed arbitrarily many times. Otherwise, after an operation has been used the maximum number of times, if a process tries to execute the operation, then the process will wait forever and will be deadlocked. Furthermore, when no operations will again be allowed to execute, it will be impossible to access the resource. Thus, there must be some set of operations such that for each there will always be some point in the future when it can be used to operate on the resource. In most circumstances, the only exceptions are initialization operations. For example, an operating system might provide an operation which is called by user processes to reserve a tape drive. If the tape drive resource may enter a state in

which the reserve operation may never again execute, then any user trying to reserve a tape drive will become deadlocked.

Definition: An operation on a shared resource is ***permanent*** if there must always be a possibility that it can execute sometime in the future.

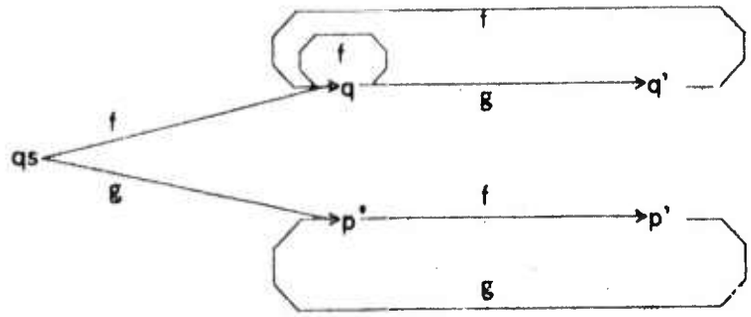
One way to specify that a set of operations can repeatedly be executed is to include them in a cycle.

Definition: A ***persistent set*** is a set of states P such that ($\forall p \in P$) ($\forall f$ which can be applied at P) $S(p,f) \in P$ and ($\forall p,q \in P$) there is an arc progression from p to q (and also one from q to p). An operation f is an ***auxiliary*** of the persistent set if ($\exists p \in P$) f may be applied at p.

Another way to describe a persistent set is that it is a smallest nonempty set of states which is closed under the successor function.

In example 2.3, {lastremove,lastinsert} forms a persistent set with auxiliary operations insert and remove. If this example is extended by adding a new initial state start and a new operation initbuf such that S(start,initbuf) = lastremove, then start isn't a member of the persistent set and initbuf isn't an auxiliary of it. If a resource only has one state as in the case where every sequence of operations is acceptable, the execution of any operation on the resource must result in that state. Therefore, it forms a persistent set and each operation is an auxiliary.

It should be noted that there may be more than one persistent set. Consider a serial resource with permanent operations f and g such that different sequences are allowed depending on whether f executes first or g does. For example, assume there are five states with qs being the initial state and S(qs,f) = S(q,f) = S(q',f) = q, S(q,g) = q', S(qs,g) = S(p',g) = p, and S(p,f) = p'.



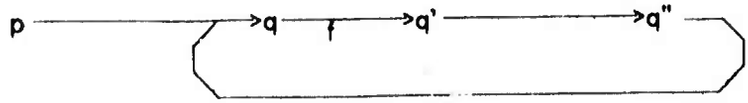
Then, {q,q'} and {p,p'} are each persistent sets.

If the state of a resource is in a persistent set, then it is easily seen that each of the auxiliaries may be executed an arbitrarily many times and that any other operations will never again be allowed to execute. Therefore, each permanent operation must be an auxiliary of every persistent set. To show that each finite state resource must have a persistent set, the following theorem is presented.

Lemma 2.7: If f is a permanent operation on a resource R, then $(\forall p)\ (\exists q)$ there is an arc progression from p to q and f may be applied at q where p and q are states of R.

Proof: Otherwise, if the state ever became p, operation f would never be able to execute again.

Theorem 2.8: If a resource R is finite state and f is a permanent operation on R, then $(\forall p)\ (\exists q,q')\ S(q,f) = q'$, there is an arc progression from p to q, and $(\forall q'')$ if there is an arc progression from q' to q", then there is an arc progression from q" to q.



Proof: By lemma 2.7, $(\exists q_0)$ such that there is an arc progression α from

p to q_0 and f can be applied at q_0. Let $S(q_0,f) = q_0'$. If for every state q" such that there is an arc progression from q_0' to q" there is an arc progression from q" to q_0, then the proof is done. Otherwise, there is a state $q_0"$ and an arc progression β from q_0' to $q_0"$ such that there is no arc progression from $q_0"$ to q_0. By lemma 2.7, there are states q_1 and q_1' such that $S(q_1,f) = q_1'$ and there is an arc progression β' from $q_0"$ to q_1. Note that $\alpha(q_0,f)\beta\beta'$ is an arc progression from p to q_1. This procedure may then be repeated. Since for $j < i$ there is an arc progression from $q_j"$ to q_i, if there is an arc progression from q_i' to q_j then there would be an arc progression from $q_j"$ to q_j, thus contradicting the assumption. Therefore, $q_j \neq q_{i+1}$ and since there are only a finite number of states, this process must eventually terminate.

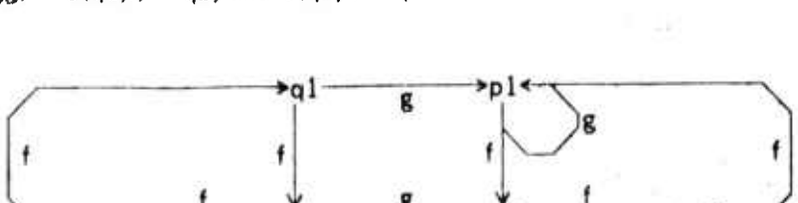
This theorem specifies a condition which must hold for the synchronization relationships. For each permanent operation, it must always be possible to enter some persistent set of which that operation is an auxiliary. Thus, it must also always be possible to enter some persistent set of which all the permanent operations are auxiliaries. If this condition doesn't hold, a deadlock can occur when some process tries to execute a permanent operation which will never again be allowed to execute.

Corollary 2.9: If a resource has at least one permanent operation, then $(\forall q)(\exists f)$ f can be applied at q.

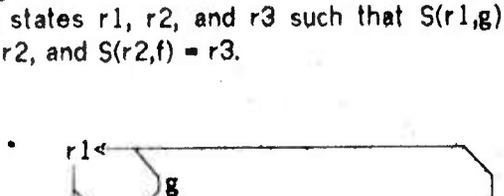
EQUIVALENT STATES

It is sometimes possible to reduce the number of states of a serial resource without changing the allowable sequences of operations.

Example 2.10: Let the synchronization for a shared resource with operations f and g be expressed by the regular expression $(g+f(g(f\ f\ f)^*g)^*(g\ f\ f\ g+f\ f))^*$. The corresponding synchronization relationships have states p1, p2, p3, q1, q2, and q3 such that S(p1,g) = S(q1,g) = S(p3,f) = p1, S(q2,g) = S(p1,f) = p2, S(p2,f) = p3, S(q3,f) = q1, S(p2,g) = S(q1,f) = q2, and S(q2,f) = q3.



The same sequences of operations may be expressed with the regular expression $(g+f\ g^*f\ f)^*$ which corresponds to the synchronization relationships with states r1, r2, and r3 such that S(r1,g) = S(r3,f) = r1, S(r2,g) = S(r1,f) = r2, and S(r2,f) = r3.



Definition: States p and q are ***equivalent*** if for every arc progression $(p,f_1)...(p_{n-1},f_n)$ there is an arc progression $(q,f_1)...(q_{n-1},f_n)$ and vice versa.

A trivial example of equivalent states p and q is when (∀f) S(p,f) = S(q,f). If α is an arc progression from S(p,f), then $(p,f)\alpha$ and $(q,f)\alpha$ are both arc progressions.

A necessary condition for a set of states to be equivalent to each other is that the same operations must be able to be applied at each of these states.

Definition: States p and q of a serial resource are ***similar*** if (∀f) f may be applied at p iff it may be applied at q.

If states p and q are similar and if whenever there are arc progressions $(p,f_1)...(p_{n-1},f_n)$ and $(q,f_1)...(q_{n-1},f_n)$ the resulting states p_n and q_n are similar, then p and q are also equivalent. This can be shown inductively on the length of the arc progressions. Since p and q are similar, there is an arc (p,f) iff there is also an arc (q,f). Assume that for n there is an arc progression $(p,f_1)...(p_{n-1},f_n)$ iff there is an arc progression $(q,f_1)...(q_{n-1},f_n)$. But the resulting states p_n and q_n are similar so there is an arc progression $(p,f_1)...(p_{n-1},f_n)(p_n,f_{n+1})$ iff $(q,f_1)...(q_{n-1},f_n)(q_n,f_{n+1})$ is also an arc progression. In addition, if p and q are equivalent and $(p,f_1)...(p_{n-1},f_n)$ and $(q,f_1)...(q_{n-1},f_n)$ are arc progressions, then $(\forall f)$ f can be applied at p_n iff it can also be applied at q_n and hence p_n and q_n are similar. Thus, states p and q are equivalent iff for any sequence of operations the corresponding arc progressions α from p to some state p' and β from q to some state q' have the property that p' and q' are similar.

To determine which states are equivalent, the set of states is first partitioned into sets of similar states. Next, taking each set of the partition which has more than one state, two states within the set are related if each operation which can execute from those states results in the same set of the partition. If the operations in the set aren't all related with each other, then the set is divided by the relation. This procedure continues until no further divisions are possible. States which remain in the same set of the partition are equivalent and can be combined.

Returning to example 2.10, the states are first partitioned as r0 = {p1,p2,q1,q2} and r3 = {p3,q3} since both f and g may execute when the resource is in any state from r0 but only f may execute when it is in a state from r3. Looking at r0, f takes p1 and q1 into r0 and p2 and q2 into r3 and g takes all four states into r0. Thus, r0 must

be divided into r1 = {p1,q1} and r2 = {p2,q2}. Now f takes p1 and q1 into r2, p2 and q2 into r3, and p3 and q3 into r1 and g takes p1 and q1 into r1 and p2 and q2 into r2. No further divisions are possible, so the new states are r1, r2, and r3 with S(r1,g) = S(r3,f) = r1, S(r2,g) = S(r1,f) = r2, and S(r2,f) = r3.

This algorithm to find equivalent states is essentially the same as that presented by Aho and Ullman [AU72, page 124] to reduce finite automata. It was necessary to modify it slightly here, though, since there are no final states in synchronization relationships and since not every operation can be applied at each state. This was done by using whether or not an operation could be applied at a state rather than whether or not the result was a final state to divide the sets of the partition. Since this algorithm can be used to reduce the number of states, it will be assumed from now on that it has been applied and that the number of states is minimal.

IMPLEMENTATION

In order for a description of the allowable sequences of operations on a resource based on the synchronization relationships to be a useful tool which can be included in a high level programming language, it must be possible to implement the relationships. A variable is used to hold the current state. Each operation contains a list of those states for which it can be applied. When a process tries to execute the operation, this list is compared with the state variable. If there is a match, the process continues by executing the operation. Otherwise, it must wait. For each operation, enough storage is needed to contain the values of the states for which the operation can be applied.

When the process starts execution, it must store the value of the state variable. It needs this value in order to calculate a new state at the completion of the operation. During execution, the state variable must be set to be the null state. This is a state at which no operation can be applied. It is used to insure that only one process at a time may execute on the resource. Thus, any attempt by a process to execute an operation on the resource while the state variable is null must fail. After execution has finished and the new state has been calculated, a search of the waiting processes is made to see if any is attempting to execute an operation which can be applied at this state. If there are any, one is selected to proceed and the state is saved. Otherwise, the state variable is set to be this new state.

A list of the processes waiting to execute on a resource is maintained so that whenever some process finishes, these can be checked. The list is ordered either by a FCFS scheme or else according to process priority. When an operation completes execution, each process in turn is checked to see if the operation it is attempting to execute may be applied at the new state. The search terminates either when one such process is found or else when the list is exhausted.

One way in which the state which results from the execution of an operation can be computed is with a table lookup. Associated with each entry in the list of states at which an operation can be applied is the resulting state. Such a scheme requires room to store a resulting state for each state at which the operation can be applied. Another possibility is to number the states in such a manner that for each operation there is some function to calculate the new state. However, there is no guarantee that such functions, if they can be found, will execute any faster than the search.

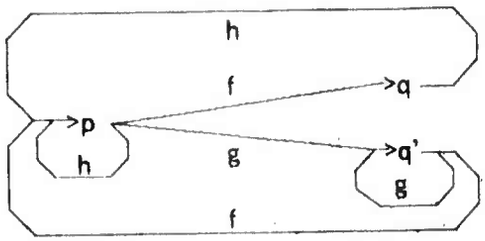
An alternative to the state variable is to use a boolean variable for each state. The boolean associated with the current state has the value TRUE and the rest have the value FALSE. The null state occurs when all of these variables are FALSE. This implementation can be made more efficient if each boolean is stored as a single bit. The state is then represented as a string of bits. For each operation, the list of states at which it can be applied can also be stored as a string of bits. The comparison between this list and the current state can be performed by ANDing the two bit strings. If the result is zero, the process must wait. A list still must be searched at the conclusion of execution in order to find the next state. However, this search will only be made once for each execution of the operation. Any checks which are made to see if the operation can be applied to the current state which fail won't result in a search.

A list of the processes waiting to execute on a resource is maintained so that whenever some process finishes, these can be checked. The list is ordered either by a FCFS scheme or else according to process priority. When an operation completes execution, each process in turn is checked to see if the operation it is attempting to execute may be applied at the new state. The search terminates either when one such process is found or else when the list is exhausted.

A modification to the waiting list is to associate a waiting list with each set of states for which some operation may be applied. Each operation will be associated with exactly one of these lists. The lists are ordered by a priority scheme as before. Now, though, the processes on top of each list are the only ones eligible to execute. There is no need to check any of the others. At the completion of execution each list

corresponding to some set of states containing the new state must be checked for waiting processes. Any process on one of these waiting lists will be able to execute. There is no need to check the list of states at which the operation it is attempting to execute can be applied. If there are any processes on these lists, one is chosen to run. In the FCFS scheme, the value of the system clock when each of the processes is put on a waiting list must be saved. This time is then used to make the selection when more than one list is checked.

Example 2.11: Let a serial resource have states p, q, and q' and operations f, g, and h such that S(p,h) = S(q,h) = S(q',f) = p, S(p,f) = q, and S(p,g) = S(q',g) = q'.



Waiting Lists

{p,q'}	{p,q}
call(f)	call(h)
call(g)	call(h)
call(g)	
call(f)	

An implementation consisting of a state variable and several waiting lists will be used. Processes which become blocked while attempting to execute the operations f and g will be put on the same waiting list since each of these operations may be applied at the set of states {p,q'}. There will also be a waiting list for processes attempting to execute h. In the diagram above, a process waiting to execute operation f is represented by the notation "call(f)". When a process tries to execute f, the state variable is checked to see if it equals either p or q'. If it does, its value is saved, it is set to the null state, and the process executes f. Otherwise, the process will be put on the waiting list for f and g. When execution completes, if the saved state is p then the new state is q and the waiting list for h is checked. If there are any processes on it,

state q is saved and one of these processes becomes unblocked and may continue execution. Otherwise, the state variable is set to be q. Likewise, if the saved state is q', then the new state is p and the waiting lists for f and g and for h are checked. If they aren't both empty, a process is chosen and state p is saved. Otherwise, the state variable is set to be p. Operations g and h are controlled similarly.

PROJECTIVE AND INJECTIVE RESOURCES

The implementation as described involves a high overhead. If only simple synchronization problems are to be handled such as the message buffer of example 2.3, many of the details of this implementation, such as the need to check more than one waiting list at the completion of an operation, aren't needed. It is useful to know what resources can be considered to be simple in this respect. This section will give an answer to that question.

There are several restrictions which can be made to an operation on a finite state resource which will result in a more efficient implementation of the operation. The first such restriction requires that an operation always results in the same state independent of the one in which it started.

Definition: An operation f is ***projective*** if $(\exists q)(\forall p)$ if f can be applied at p then $S(p,f) = q$. A finite state serial resource is ***projective*** if every operation on it is projective.

In the message buffer of example 2.3, remove always results in lastremove and insert always results in lastInsert. Therefore, the message buffer is a projective resource.

Example 2.12: Let the regular expression $(ff^*g)^*$ represent the allowable

sequences of operations on a shared resource. The synchronization relationships consist of two states p and q where p is the initial state, S(p,f) = S(q,f) = q, and S(q,g) = p. Since f always results in state q and g always results in state p, each is projective.

The state of a projective resource represents only the most recent operation to execute on that resource since each operation forgets whatever history was contained in the previous state. This implies that there may at most be one more state than there are operations, an initial state and a state corresponding to each operation.

To show that for a projective resource there can't be two similar states p and q, let f be any operation which can be applied at p. Then f can also be applied at q. But since f is projective, S(p,f) = S(q,f). Therefore, p and q are equivalent, which is a contradiction of the assumption that no two states of a finite state resource are equivalent.

Since each projective operation always results in the same state, this resulting state is no longer a function of the state from which the operation started. The implementation can therefore be made simpler since the resulting state doesn't need to be calculated but is a constant. Also, there is no longer any need for an operation to remember what the state was when it started.

Another restriction which can be made to a finite state resource is to require that, with respect to each operation, the successor function S is one to one.

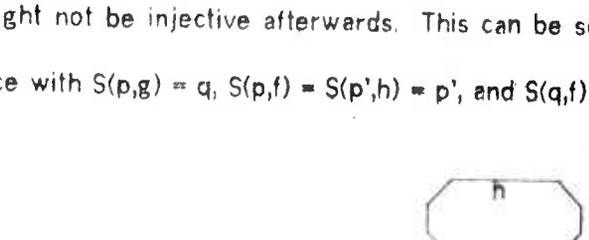
Definition: An operation f is *injective* if (∀q) there is at most one state p such that S(p,f) = q. A finite state serial resource is *injective* if every operation on it is injective.

Thus, if two different arcs result in the same state then they must have different operations.

Example 2.13: If the regular expression $(fg+gf)^*$ represents the sequences in which the operations of a shared resource are allowed to execute, then the corresponding synchronization relationships consist of an initial state p and states q and q' such that S(p,f) = q, S(p,g) = q', and S(q,g) = S(q',f) = p.

The resource is injective since p is the only state such that S(p,f) = q and S(p,g) = q', q is the only state such that S(q,g) = p, q' is the only state such that S(q',f) = p, and there is no state p' such that S(p',f) = q' or S(p',g) = q. However, it isn't projective since neither f nor g is a projective operation.

If a serial resource is projective before the equivalent states are combined, then it must also be projective afterward. This is trivially true since if an operation may only result in one state and then states are combined it still will only be able to result in one state. However, a serial resource which is injective before equivalent states are combined might not be injective afterwards. This can be seen by considering the injective resource with S(p,g) = q, S(p,f) = S(p',h) = p', and S(q,f) = S(q',h) = q'.



States p' and q' are equivalent. Combining them into a new state p" yields S(p,f) = S(q,f) = p". Therefore, the resource is no longer injective.

The process of combining equivalent states can sometimes be reversed to make an operation which isn't injective into one that is. Assume that $S(p,f) = S(p',f) = q$. If there is no arc progression from q to either p or p', then create a new state q_i' for every state q_i to which there is an arc progression from q. Also, create a new state q'. For each q_i' and operation g, if $S(q_i,g) = q_j$ then let $S(q_i',g) = q_j'$. Also, let $S(p',f) = q'$. If there was an arc progression from q to p, then a state p" would have been created such that $S(p",f) = q'$ and there would have been an arc progression from q' to p". This procedure would then have continued indefinitely without f ever becoming injective.

Example 2.13 shows that not every injective resource is projective. On the other hand, the projective resource of example 2.12 isn't injective since $S(p,f) = S(q,f) = q$. The intersection of these two serial resource classes, though, turns out to be an interesting class itself.

Definition: An operation is ***simple serial*** if it is both projective and injective. A resource is ***simple serial*** if every operation on it is simple serial (it is both a projective and an injective resource).

For each operation of a simple serial resource, there is only one state from which it may start execution and only one state which can result. It is easily seen that the message buffer of example 2.3 is such a resource.

If the several waiting list implementation is used for a simple serial resource, each list needs to be associated with only one state. This is because each operation may only be applied at one state. This means that at the completion of execution, an operation will only check one list to see if any processes waiting can now continue.

The boolean state variables can also be considered to be boolean semaphores. The result is that each operation starts execution by doing a P on one of these semaphores and concludes by doing a V on the semaphore associated with the resulting state.

If the sequences of operations defined on a serial resource are controlled by preceding each operation with one P and following it with one V, then each semaphore must be boolean. This is because otherwise if a semaphore ever attained a value of more than one, any operation which started with a P on that semaphore would be able to execute in parallel with itself. Also, only one semaphore can have a positive value when no operation is executing and none can have a positive value when one process is executing on the resource. Thus, each semaphore may be thought of as a state and for each operation the semaphore on which a P is done represents the state that the operation waits for and the semaphore on which a V is done represents the resulting state. Therefore, the class of serial resources which can be implemented with each operation preceded by one P and followed by one V is the same as the simple serial resources.

PRIORITY

When more than one waiting process can start execution from a state which results from the currently executing process, a choice must be made. The decision criteria is referred to as a priority policy. One such possible policy, FCFS, chooses the process which has been waiting the longest. The waiting lists act like simple queues in this case. Another possible policy is to use the same priority for each process that the scheduler does. The decision as to which priority policy should be used is the responsibility of the system designer.

A warning must be made about the possibility of starvation when a policy other than FCFS is used. This can occur if for some state more than one process can start execution whenever the resource enters that state. If one of these waiting processes has a sufficiently low priority, it might never be chosen. This problem doesn't occur with a FCFS policy since the longer a process waits, the higher its priority gets.

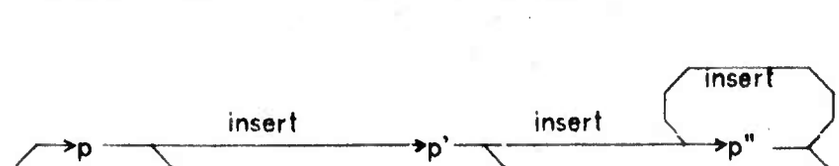
Often when more than one operation may be applied at a given state, it is necessary to give processes waiting to execute some operations a higher priority than processes waiting to execute the others.

Example 2.14: Consider again the message buffer of example 2.1 with the modification that any sequence of the operations insert and remove are acceptable. In order that the most current message is received, insert will have priority over remove.

Another example is a storage allocator on which the operations getspace and releasespace are defined. Releasespace has the side effect that it will collapse any two adjacent blocks of free storage into one. Therefore, it will have priority over getspace.

The priority relation among the operations for a state must form a partial ordering. This means that for operations f and g, exactly one of the following is true. Either f has priority over g, g has priority over f, or they have equal priority. In addition, this relation must be transitive. This means that if f has priority over g and g has priority over h then f also has priority over h. However, since the operation priority is defined for each state, it is possible that the partial ordering between two operations is different for the various states at which they each may be applied. This may be done to prevent starvation. For instance, consider example 2.14 again. After

the buffer has been written twice, processes trying to receive information from it will be given a chance. Three states are needed with S(p,insert) = p', S(p',insert) = S(p",insert) = p", and S(p,remove) = S(p',remove) = S(p",remove) = p.



Operation insert is given priority at p and p' and remove is given priority at p".

It can be shown that p, p', and p" are equivalent. They can't be reduced, though, due to the priority differences. The algorithm described above to find equivalent states must therefore be modified to handle priority. An initial partitioning of similar states is made as before. For each set created by the partitioning, the operations which can be applied at the states of the set must have the same relative priority at each of those states. If they don't, then that set must be divided. After this step in the example, the partition would be {p,p'} and {p"}. The rest of the algorithm is then applied.

The implementation of operation priority is simplest when the several waiting list policy is used. If operations f and g may be applied at some state with f having the higher priority, then when the resource enters that state the waiting list for processes trying to execute f is checked. Only if this list is empty is the one with processes trying to execute g considered. A problem arises if f and g may be applied at exactly the same set of states. Then processes trying to execute these operations wait on the same list. This rule must be altered whenever one of these operations has

priority over the other at any of these states where they can be applied. In that case, the waiting list must be divided.

In the single waiting list implementation, the processes are ordered according to which operations they wait on. However, problems arise when a partial ordering of the operations can't be made. This can occur when one operation has priority over a second at some state but the priority is reversed (or they both have the same priority) at another state. Another case is when the transitive law doesn't hold. An example is when an operation f has priority over an operation g at one state, g has priority over h at a second state, and h has priority over f at a third. When such a situation occurs, the entire waiting list might have to be searched for each priority class.

CHAPTER III

SUBCLASSES OF REGULAR EXPRESSIONS

As seen in the last chapter, the synchronization for any finite state serial resource can be expresed using a regular expression. However, it may be desirable to restrict the serial resources in a programming system to simple serial or else to some other subclass of finite state. If the synchronization is expressed using a regular expression, a test is necessary to make sure that the synchronization expressed is in the subclass. Alternatively, if regular expressions can be restricted correspondingly, no such test would be necessary. Several such subclasses will be suggested. In addition, a comparison will be made between one such subclass and simple serial resources. First, though, it will be necessary to look at the relationship between regular expressions and finite automata further.

As Hopcroft and Ullman [HU69] point out, a regular expression R can be converted to a corresponding nondeterministic finite automaton in the following manner. The finite automaton $(\{p,q,q'\},\{f\},\delta,p,\{q\})$ where $\delta(p,f) = q$ and $\delta(q,f) = \delta(q',f) = q'$ corresponds to the regular expression with the single symbol f. Note that q' is a dead state since it isn't a final state and since once the automaton enters q' it will never exit from it.

If the finite automaton $M = (K,\Sigma,\delta,p,F)$ corresponds to the regular expression R, then the finite automaton $(K\cup\{p'\},\Sigma,\delta',p',F\cup\{p'\})$ where

$$\begin{array}{lll} \delta'(p',f) & = \{\delta(p,f),p\} & \text{if } \delta(p,f) \in F \\ & = \delta(p,f) & \text{otherwise} \\ \delta'(q,f) & = \{\delta(q,f),p\} & \text{if } \delta(q,f) \in F \\ & = \delta(q,f) & \text{otherwise } (\forall q \in K) \end{array}$$

corresponds to the regular expression R^*. If $p \in F$ and $\delta(q,f) = p$ then $\delta'(q,f) = \{\delta(q,f),p\} = \{p\} = \delta(q,f)$. Therefore, F may be replaced by $F' = F-\{p\}$ in the definition of δ'. Since $(\forall q' \in F')\ (\forall q \in K \cup \{p'\})\ (\forall f \in \Sigma)$ if $q' \in \delta'(q,f)$ then $p \in \delta'(q,f)$, a new final state q" can be created to replace {q',p} and q' can be deleted. The states which can result from reading a symbol when the state is q" must be the same as those which can result when the automaton is in either state q' or state p. Therefore, $\delta'(q'',f) = \{\delta'(q',f),\delta'(p,f)\}$. By renaming each new state q" representing {q',p} to be q', the finite automaton is changed so that

$$\begin{array}{lll} \delta'(p',f) & = \delta(p,f) & \\ \delta'(q,f) & = \{\delta(q,f),\delta(p,f)\} & (\forall q \in F-\{p\}) \\ & = \delta(q,f) & \text{otherwise} \end{array}$$

Notice that $(\forall f \in \Sigma)\ \delta'(p,f) = \delta'(p',f)$ and therefore p and p' are equivalent. Thus, if $p \in F$ they can be combined. Otherwise, if nothing can result in p then it can be deleted. In either of these cases, the initial state can be renamed to be p and the resulting finite automaton is (K,Σ,δ',p,F) where $\delta'(p',f)$ is no longer defined.

Let $M = (K,\Sigma,\delta,p,F)$ and $M' = (K',\Sigma',\delta',p',F')$ be finite automata corresponding to regular expressions R and R' such that $K \cap K'$ is empty. For f not in Σ define $(\forall q \in K)$ $\delta(q,f)$ to be a dead state and for g not in Σ' define $(\forall q \in K')$ $\delta'(q,g)$ to be a dead state. The finite automaton $M''_1 = (K \cup K', \Sigma \cup \Sigma', \delta''_1, p, F''_1)$ where

$$\begin{array}{lll} \delta''_1(q,f) & = \delta(q,f) & (\forall q \in K-F) \\ & = \{\delta(q,f),\delta'(p',f)\} & (\forall q \in F) \\ & = \delta'(q,f) & (\forall q \in K') \end{array}$$

and where $F''_1 = F \cup F'$ if $p' \in F'$ or $F''_1 = F'$ otherwise corresponds to the regular expression RR'. The finite automaton $M''_2 = (K \cup K' \cup \{p''\}, \Sigma \cup \Sigma', \delta''_2, p'', F''_2)$ where

$$\begin{aligned} \delta''_2(p'',f) &= \{\delta(p,f), \delta'(p',f)\} \\ \delta''_2(q,f) &= \delta(q,f) \qquad (\forall q \in K) \\ &= \delta'(q,f) \qquad (\forall q \in K') \end{aligned}$$

and where

$$\begin{aligned} F''_2 &= F \cup F' \cup \{p''\} \text{ if } p \in F \text{ or } p' \in F' \\ &= F \cup F' \qquad \text{otherwise} \end{aligned}$$

corresponds to the regular expression R+R'. If $\Sigma \cap \Sigma'$ is empty and δ and δ' are deterministic, then so are δ''_1 and δ''_2 since $(\forall q \in K)\ (\forall f \in \Sigma \cup \Sigma')$ either f isn't in Σ and $\delta(q,f)$ is dead or else f isn't in Σ' and $\delta'(p',f)$ is dead.

RESTRICTED REGULAR EXPRESSIONS

As has already been shown, the desired sequencing of operations on any finite state serial resource may be expressed using a regular expression. Since for implementation reasons a system designer might wish to restrict himself to simple serial resources, it would be helpful to know what subclass of regular expressions provides exactly the synchronization needed for these resources. In an attempt to do this, the synchronization provided by several subclasses will be examined.

Definition: An ***initial loop*** regular expression is defined recursively as follows. A regular expression R^* is initial loop. RR' is initial loop if R is and R+R' is initial loop if either R or R' is. No other regular expression is initial loop. A ***final loop*** regular expression is defined similarly. A regular expression R^* is final loop. R+R' is selection final loop if either R or R' is final loop and RR' is (selection) final loop if R' is.

Some examples of initial loop regular expressions are f^*, f^*g, and f^*g+h. The regular expressions f^*, fg^*, and fg^*+h are final loop.

Definition: A *restricted* regular expression is also defined recursively. A single symbol regular expression is restricted. If R is restricted and is neither initial nor final loop then R^* is restricted. If R and R' are restricted and have no symbols in common, then R+R' is restricted if it isn't initial loop and RR' is restricted if either R isn't final loop or else it isn't selection final loop and R' isn't initial loop.

The general requirements for a regular expression to be restricted are that no symbol may be used more than once and that subexpressions of the form R^* must occur in the context $R'R^*R''$ where R' isn't final loop and R" isn't initial loop. The exceptions are that R" may be omitted provided that nothing else may follow R^* and the whole expression may take the form R^*.

To help understand which regular expressions are being excluded, consider fgf, $(fg^*)^*$, f^*g+h, f^*g^*, and $(fg^*+h)e$. None of these is a restricted regular expression. In the first, the symbol f is repeated twice. In the second, $R = (fg^*)$ is final loop and therefore R^* isn't restricted. In the third, f^*g+h is of the form R+R' and is initial loop so it isn't restricted. The next violates the condition of a final loop subexpression being followed by an initial loop subexpression. Finally, fg^* is final loop and therefore (fg^*+h) is selection final loop and can't be followed by anything. Regular expressions of the form $(R^*)^*$ aren't restricted since R^* is initial loop. However, the same sequence of symbols can be represented by the restricted regular expression R^*.

As might be expected, the rules for constructing a finite automaton from a restricted regular expression can be simplified. In addition, several interesting properties are true of the finite automata so constructed.

Theorem 3.1: For a restricted regular expression R with the corresponding finite automaton (K,Σ,δ,p,F) the following properties are true.

Property 3.1.1: Either R is final loop or $(\forall q \in F)\ (\forall f \in \Sigma)\ \delta(q,f)$ is dead.

Property 3.1.2: $(\forall f \in \Sigma)\ (\forall q,q' \in K)$ either $\delta(q,f)$ or $\delta(q',f)$ is dead.

Property 3.1.3: $(\exists f \in \Sigma)\ \delta(p,f)$ isn't a dead state.

Property 3.1.4: Either R is initial loop or $(\forall q \in K)\ (\forall f \in \Sigma)\ \delta(q,f) \neq p$ and p isn't in F.

Property 3.1.5: If R is simple (not selection) final loop, then there is only one state in F.

Furthermore, let (K,Σ,δ,p,F) be the finite automaton corresponding to R and $(K',\Sigma',\delta',p',F')$ correspond to R'. Then $(K\text{-}F,\Sigma,\delta_1,p,\{p\})$ corresponds to R^*, either $((K\text{-}F)\cup K',\Sigma\cup\Sigma',\delta_2,p,F')$ corresponds to RR' or there is only one state $p'' \in F$ and $(K\cup(K'\text{-}\{p'\}),\Sigma\cup\Sigma',\delta_3,p,F')$ corresponds to RR', and $(K\cup(K'\text{-}\{p'\}),\Sigma\cup\Sigma',\delta_4,p,F\cup F')$ corresponds to R+R' where

$\delta_1(q,f)$	$= p$	if $\delta(q,f) \in F$
	$= \delta(q,f)$	otherwise $(\forall q \in (K\text{-}F))$
$\delta_2(q,f)$	$= p'$	if $\delta(q,f) \in F$
	$= \delta(q,f)$	otherwise $(\forall q \in (K\text{-}F))\ (\forall f \in \Sigma)$
	$= \delta'(q,f)$	$(\forall q \in K')\ (\forall f \in \Sigma')$
$\delta_3(p'',f)$	$= \delta'(p',f)$	$(\forall f \in \Sigma')$
$\delta_3(q,f)$	$= \delta(q,f)$	$(\forall q \in K)\ (\forall f \in \Sigma)$
	$= \delta'(q,f)$	$(\forall q \in (K'\text{-}\{p'\}))\ (\forall f \in \Sigma')$
$\delta_4(p,f)$	$= \delta'(p',f)$	$(\forall f \in \Sigma')$
$\delta_4(q,f)$	$= \delta(q,f)$	$(\forall q \in K)\ (\forall f \in \Sigma)$
	$= \delta'(q,f)$	$(\forall q \in (K'\text{-}\{p'\}))\ (\forall f \in \Sigma')$

Any arguments for which δ_2, δ_3, or δ_4 are undefined are dead.

Proof: The proof is based on the invariance of the properties over the construction of the finite automaton. The details are presented in the Appendix.

Notice that for the finite automaton constructed in this manner from a restricted

regular expression, δ is deterministic. Also, for each symbol f there is at most one state q such that $\delta(q,f)$ isn't dead. This means that for each operation of the corresponding set of synchronization relationships there is at most one state at which it can be applied. Therefore, the corresponding resource must be simple serial. In addition, the only states which can be equivalent are those for which no operations may be applied. The result is that the algorithms to make the synchronization relationships deterministic and to remove equivalent states aren't needed. All that need be done is to combine all of the states at which no operations can be applied.

Z EXPRESSIONS

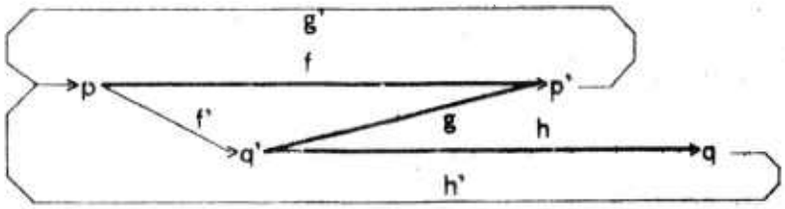
Next, the relationship between restricted regular expressions and synchronization relationships will be examined. It will be shown that if the synchronization for a shared resource can be expressed using a restricted regular expression then the resource must be simple serial. However, there are some simple serial resources for which the synchronization can't be expressed using a restricted regular expression.

In order to characterize those synchronization relationships for simple serial resources which can't be written as restricted regular expressions, it will be necessary to study groups of three arc progressions such that the first and second have the same final state and the second and third have the same initial state. It will be necessary to require that any given state may occur in at most two of these arc progressions. However, there is no requirement that the first or third can't be null. By a null arc progression is meant one from a state to itself which contains no arcs.

Definition: A *Z expression* from a state p to a state q consists of arc progressions α from p to some state q_n, $\beta = (q_0,f_1)\ldots(q_{n-1},f_n)$ from some state q_0 to q_n, and γ from q_0 to q such that $(\forall i, 0<i<n)$ $q_i \neq p$, $(\forall j, 0<j\leq n)$ $q_j \neq q$, $(\forall i, 0<i<n)$ there are not two arcs (q_i,f) in α and (q_j,g) in γ, there is no arc (q_0,f) in α, and there is no arc (q_n,f) in γ.

Several conditions which must be true of Z expressions but which aren't explicitly stated may be derived from this definition. One is that $q_0 \neq q_n$. Otherwise, either γ is empty and $q = q_0 = q_n$ or else $(q_n,f) = (q_0,f)$ is in γ for some symbol f from the input alphabet. Another is that $q_0 \neq p$. Otherwise, either α is empty and $q_n = p = q_0$, violating the above condition, or else there is an arc $(p,f) = (q_0,f)$ in α. Finally, if $p = q$ then $q_0 \neq q$ and $q_n \neq p$. If this wasn't true, then $p = q_0$ or $q = q_n$. Thus, neither α nor γ can be empty when $p = q$.

As an example, consider the synchronization relationships with states p, p', q', and q and operations f, f', g, g', h, and h' such that S(p,f) = S(q',g') = p', S(p,f') = q', S(q',h) = q, and S(q,h') = S(p',g') = p.



Then the arcs (p,f), (q',g), and (q',h) form a Z expression from p to q. Also, the arc (q',g) forms a Z expression from p' to q' for which the α and γ arc progressions are each empty.

In what follows, it will sometimes be easier to deal with Z expressions restricted such that α and β have only their final states in common and β and γ have only their

initial states in common. It will also be required that p may not occur in α other than at the start and q may not occur in γ other than at the end.

Definition: A *simple Z expression* from a state p to a state q consists of a Z expression $\alpha = (p_0,g_1)...(p_{m-1},g_m)$ $\beta = (q_0,f_1)...(q_{n-1},f_n)$, and γ where α is from $p = p_0$ to q_n and γ is from q_0 to q such that $(\forall i, 0<i<n)$ there is no arc (q_i,f) in α or γ, $(\forall j, 0<j<m)$ $p \neq p_j$, and there is no arc (q,f) in γ.

Actually, the use of simple Z expressions isn't really a restriction since every Z expression may be reduced to a simple Z expression.

Lemma 3.2: If there is a Z expression from a state p to a state q, then there also is a simple Z expression from p to q.

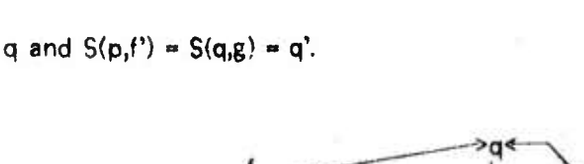
Proof: Let $\alpha = (p_0,g_1)...(p_{m-1},g_m)$, $\beta = (q_0,f_1)...(q_{n-1},f_n)$, and γ be a Z expression from $p = p_0$ to q. If $(\exists j, 0<j<m)$ $p_j = p$, then $(p_j,g_{j+1})...(p_{m-1},g_m)$, β, and γ form a Z expression from p to q. If there is an arc (q,f) in γ, then γ can be written as $\gamma'(q,f)\gamma''$ where γ' is a (possibly empty) arc progression which doesn't contain such an arc and α, β, and γ' form a Z expression from p to q. If $(\exists i, 0<i<n)$ there is an arc (q_i,f) in α, then $(\exists j, 0<j<m)$ $q_i = p_j$. By the definition of a Z expression, there is no arc (q_i,g) in γ so $(p_0,g_1)...(p_{j-1},g_j)$, $(q_0,f_1)...(q_{i-1},f_i)$, and γ form a Z expression from p to q. Likewise, if $(\exists i, 0<i<n)$ there is an arc (q_i,f) in γ, then γ can be written as $\gamma'(q_i,f)\gamma''$ where γ'' is from q_i to q. By the definition of a Z expression, there is no arc (q_i,f) in α so α, $(q_i,f_{i+1})...(q_{n-1},f_n)$, and γ'' form a Z expression from p to q.

PERSISTENT SET ENTRY STATES

Of particular interest will be Z expressions from the initial state to what may be regarded as the final states. In the conversions from a restricted regular expression to a finite automaton, it may be seen that the only final state of a loop was its initial state and that either the regular expression was final loop or else any final states had no nondead successors.

Definition: An *entry state* of a persistent set is an element q of the persistent set such that either q is the initial state of the resource or ($\exists$q', q' not an element of the persistent set) ($\exists$g) S(q',g) = q.

Let the synchronization for a serial resource be expressed by the regular expression $(f+f'h)(gh)^*$. The synchronization relationships have three states p, q, and q' such that S(p,f) = S(q',h) = q and S(p,f') = S(q,g) = q'.



The states q and q' form a persistent set with g and h being the auxiliary operations. Since p isn't in the persistent set, S(p,f) = q, and S(p,f') = q', both q and q' are entry states into the persistent set. Notice that (p,f), (q',h), and ϵ form a Z expression from p to q' and (p,f'), (q,g), and ϵ form a Z expression from p to q. The presence of these Z expressions can also be deduced from the following result

Lemma 3.3: If some persistent set has more than one entry state, then the initial state of the resource p isn't in this set and there is a Z expression from p to each of these entry states.

Proof: If p is an element of the persistent set, then every state must also be in the persistent set and it can be the only entry state. Otherwise, let q and q' be entry states for the persistent set. There must be arc progressions α from p to q, α' from p to q', γ from q' to q, and γ' from q to q'. Then α and γ form a Z expression from p to q' and α' and γ' form a Z expression from p to q.

The final states of a restricted regular expression can now be characterized.

Lemma 3.4: If the synchronization for a resource can be expressed with a restricted regular expression, then the set of persistent set entry states and states with no nondead successors is the same as the set of final states produced using the construction in theorem 3.1.

Proof: It will also be shown that there must be an arc progression from every state to a final state. The proof is by induction on the complexity of the regular expression. For a single element regular expression this is certainly true. Assume that it is true for R. Since there is an arc progression from every state to a final state, there must be an arc progression from every state to the initial state p in R^*. Thus, all of the states form a persistent set and there are no states such that every successor is dead. Since p is the only final state, the lemma is true for R^*. Assume that it is true for R and R'. For RR' and δ_2 since there must be an arc progression from every state of K-F to a state of F in R, there must be an arc progression from every state of K-F to p' in RR'. Also, there is no arc progression from any state of K' (including p') to a state of K. Thus, every state of K-F has a nondead successor and none can be in any

persistent set. Since $(\forall q \in K')\ \delta_2(q,f) = \delta'(q,f)$, every successor of q is dead in RR' iff they all are dead in R'. Also, p' is the only state in K' such that $\delta_2(q,f) = p'$ for a state $q \in K-F$. Thus, a state $q \in K'$ is a persistent set entry state in RR' iff it also is in R'. Since the final states for RR' is F' and since there is an arc progression from every state of K' (including p') to some state of F', the lemma and hypothesis are true for RR'. If δ_3 is used, then there can only be one state q in F. Thus, q is the only state of K such that $\delta_3(q,f) \in K'$. Since there is an arc progression from p' to every element of F' in R', there must also be an arc progression from q, and therefore from every element of K, to every element of F'. Thus, no state of K has all dead successors. As with δ_2, there is no $q \in K'-\{p'\}$ such that $\delta_3(q,f) \in K$ for some f and also $(\forall q \in K'-\{p'\})\ \delta_3(q,f) = \delta'(q,f)$. Thus, no state of K can be in a persistent set and a state of $K'-\{p'\}$ is a persistent set entry state or has no nondead successors in RR' iff the same is true in R'. Since the final states of RR' are F', the lemma and hypothesis must be true for RR'. Finally, for R+R', since there must be an arc progression from every state in R to a state in F and there must also be an arc progression from every state of R' to a state in F', the hypothesis will be true in R+R'. For every state $q \in (K-\{p\})\cup(K'-\{p'\})$ an operation may be applied at q in R+R' iff it could be applied at q in R or in R' and the resulting state will be the same. Also, by properties 3.1.3 and 3.1.4, p and p' have at least one nondead successor and no arc results in these states in R and R' and the same is true for p in R+R'. Therefore, a state will have no nondead successors or be a persistent set entry state in R+R' iff the same was true in either R or in R'.

SYNCHRONIZATION AND RESTRICTED REGULAR EXPRESSIONS

It can now be shown that each restricted regular expression describes the allowable sequences of operations for some simple shared resource such that in the synchronization relationships there is no Z expression from the initial state to any state q such that either no operation may be applied at q or else q is a persistent set entry state.

Theorem 3.5: A shared resource on which the allowable sequences of operations are given by a restricted regular expression is simple serial with no Z expression from the initial state to a final state.

Proof: The lack of a Z expression from the initial state to a final state is invariant over the construction of the finite automaton. The details are presented in the Appendix.

Corollary 3.6: An elementary path expression without curly brackets is simple serial and contains no Z expression from the initial state to itself.

This last theorem shows that every resource for which the allowable sequences of operations can be given by a restricted regular expression is simple serial but that not every simple serial resource can have the synchronization for it expressed in this manner. The next question is whether or not the synchronization for every simple serial resource with none of these Z expressions can even be expressed using restricted regular expressions.

Theorem 3.7: A simple serial resource with no Z expression from the initial state to a state q such that either no operation may be applied at it or else q is a persistent set entry state can be written as a restricted regular expression without repeated names.

Proof: The proof shows that the synchronization relationships can be split into nonempty parts reversing the construction from a restricted regular expression or else a loop can be broken if there are no Z expressions. The details are presented in the Appendix.

It has been shown that there are some simple serial resources for which the synchronization can't be given using restricted regular expressions. Perhaps allowing operation names to be repeated would help to solve this problem. Unfortunately, this is not the case.

Theorem 3.8: The synchronization for any finite state resource may be described using a regular expression in which the conditions for a restricted regular expression hold but in which operation names may be repeated.

Proof: It will be shown that for every regular expression R there is a regular expression R' such that R and either R' or (R'+ϵ) express the same strings and the conditions for a restricted regular expression hold where ϵ is the null expression. Since whether or not the null string is acceptable is unimportant when expressing the synchronization of operations on a resource, R' satisfies the theorem. The proof will be by induction on the complexity of the expression. Clearly, a single symbol expression is a restricted regular expression which is neither initial nor final loop. Assume that R and R' satisfy the conditions and are neither initial nor final loop. Then R+R', RR'+R, RR'+R', RR'+R+R', and RR*R+R all satisfy the conditions and none is either initial nor final loop. Since $R^* = (R+\epsilon)^* = (RR^*R+R)+\epsilon$, $(R+\epsilon)+R' = R+(R'+\epsilon) = (R+\epsilon)+(R'+\epsilon) = (R+R')+\epsilon$, $R(R'+\epsilon) = RR'+R$, $(R+\epsilon)R' = RR'+R'$, and $(R+\epsilon)(R'+\epsilon) = (RR'+R+R')+\epsilon$ the theorem is proved.

Another change which can be made is to remove the conditions but to continue to prohibit the repeating of operation names.

Definition: A ***nonrepeat regular expression*** is a regular expression in which subexpressions of the form $R^+ = RR^*$ and $R+\epsilon$, where ϵ is the null subexpression, are allowed but in which no operation name is repeated.

The symbol ϵ may be simulated by creating a null operation f which will never be called. Then f^* is the same as the symbol ϵ.

Lemma 3.9: For a restricted regular expression, the initial state of a final loop must be a final state.

Proof: The proof is by induction. If the regular expression is of the form R^*, then by theorem 3.1, the initial state is a final state. If the regular expression is of the form RR', then the final states of R' are final states. Since RR' is final loop iff R' is, if the lemma holds for R', then it holds for RR'. Likewise, if the expression is of the form R+R', then the final states are those of R and R'. Also, R+R' is final loop iff either R or R' is. Thus, if the lemma holds for R and R', then it holds for R+R'.

Theorem 3.10: A serial resource on which the allowable sequences of operations is given by a nonrepeat regular expression either isn't simple serial or else the synchronization can be expressed using a restricted regular expression.

Proof: If a nonrepeat regular expression isn't restricted, then one of the following situations must be true.

Case 1: A subexpression has the form R^* and R is restricted and simple final loop. If R has the form R'^*, then R and R^* are equivalent so the subexpression could have been written as R. Assume that R has the form $R'R''^*$. Since R is restricted, R' can't be final loop and by theorem 3.1, properties 3.1.1 and 3.1.3, its initial state p can't be

one of its final states. Thus, no operation of R" can be applied at p in R and since p is also the initial state of R, none can be applied at the initial state of R^*. However, after some string of R' is executed in R^*, any operation which can be applied at either the initial state of R' or the initial state of R" can be applied. Thus, this state p' can't be equivalent to the initial state. Any operation which can be applied at p in R' can be applied at the initial state of R^* and at p'. Therefore, R^* can't be simple serial.

Case 2: A subexpression has the form R^* and R is restricted and selection final loop. There must be a subexpression of R of the form $R_1R_2^* + R_3$ where R_1 is neither initial nor final loop. Thus, no operation which can be applied at the initial state of R_2^* can be applied at the initial state of R. By lemma 3.9, the initial state of R_2^* must be a final state of R. Therefore, there must be a state p' in R^* at which everything which may be applied at either the initial state p of R or at the initial state of R_2^* may be applied. By property 3.1.3 of theorem 3.1, p and p' can't be equivalent but everything which may be applied at the initial state of R may be applied at both states. Thus, R^* isn't simple serial.

Case 3: A subexpression has the form R^* and R is restricted and initial loop but not final loop. Thus, R has the form R'^*R'' where R" is neither initial nor final loop and its initial state can't be a final state. If no operation other than those contained in R can be applied to the final state of R^*, then R^* can be written as $(R'+R'')^*$, which is restricted. Assume that operation f can be applied at the final states of R^*. At the initial state of R^*, f may be applied along with any operations which may be applied at the initial states of R' and R". However, if a string from R' executes, only those operation which may be applied at the initial states of R' or R" may execute. Therefore, there are two distinct states at which these operations may execute and R^* isn't simple serial.

Case 4: A subexpression has the form R+R', R is initial loop, and both R and R' are restricted. Thus, R has the form $R_1^*R_2$ (R_2 is optional). Any operation which may be applied at the initial states of R_1 and R' may be applied at the initial state of R+R'. However, after a string of operations from R_1 have executed, the operations which may be applied at the initial state of R_1 may be applied but those from R' can't be. Thus, R+R' isn't simple serial.

Case 5: A subexpression has the form RR' where R is final loop, R' is initial loop, and both R and R' are restricted. Thus, by lemma 3.9, there is a loop R_1^* in R such that the initial state of R_1^* is a final state of R. Also, R' has the form R_2^*R''. Let p be the initial state of R_1 in RR'. Any operation which may be applied at either p in R_1 or at the initial state of R_2 may be applied at p in RR'. However, after a string of operations from R_2 has executed, those operations which

may be applied at the initial state of R_2 may be applied but those which can be applied at p in R can't be. Therefore, RR' isn't simple serial.

Case 6: A subexpression has the form RR' where R is selection final loop and R and R' are restricted. As with case 2, R must have a subexpression of the form $(R_1R_2^* + R_3)$ and the initial state of R_2^* and the final state of R_3 must be final states of R. Any operation which can be applied at the initial state of R' can be applied at both of these final states in RR'. The operations which can be applied at the initial state of R_2 in R can't be applied at the final state of R_3, however. Thus, RR' isn't simple serial.

Case 7: There is a subexpression of the form R^+ where R is restricted. If no operation not in R can be applied at a final state of R^+, then R^+ and R^* are the same for synchronization purposes. Assume that f isn't in R but can be applied at a final state of R^+. It can't be applied at the initial state of R^+, but it can be applied after some sequence of R. Thus, there are two different states at which initial operations of R can be applied and R^+ isn't simple serial.

Case 8: There is a subexpression of the form ϵ. Since $R\epsilon = R = \epsilon R$, $\epsilon^* = \epsilon$, and $(R+\epsilon)^* = R^*$, assume that ϵ is included in a subexpression of the form $(R+\epsilon)$. If the initial state of R is a final state or if no operation not in R can be applied at the final states of R, then $(R+\epsilon) = R$. Assume that R is simple serial, the initial state p of R isn't a final state, and there is at least one operation f not in R which can be applied at the final states of $(R+\epsilon)$. If g is an operation of R which can be applied at its initial state, then it can't be applied at any other states, including the final states. However, f can be applied at both the initial and final states of R. Therefore, $(R+\epsilon)$ isn't simple serial.

Thus, no nonrepeat regular expression describes the synchronization for a simple serial resource which can't be described using a restricted regular expression.

RELATIONSHIP TO CONTROL STRUCTURES

As shown in chapter II, the synchronization relationships for a simple serial resource may be thought of as a directed graph with each state represented by a

node and each operation represented by an arc. This graph has the property that there are arc progressions from the the node representing the initial state to each of the other nodes. Flowcharts with the property that each arc represents a different computation with one entry and one exit point are also equivalent to the same set of directed graphs.

Regular expressions and control structures from programming languages can also be compared. The expression RR' means first R and then R' must occur. Likewise, concatenating two computations means do the first and then do the second. The expression R^* means that R occurs zero or more times and the statement WHILE p DO R means that R will be executed zero or more times. The expression R+R' means that either R or R' must occur and the statement IF p THEN R ELSE R' means that either R or R' will be executed. Since

DO R UNTIL p = R; WHILE NOT p DO R

and $R^+ = RR^*$, they each produce the same sequences. Finally, R+ϵ means that R may optionally occur and IF p THEN R means that R will optionally be executed.

The results that are given above about the relationship between regular expressions and simple serial resources can be applied to flowcharts in which each arc represents a different computation and programs which are written using the above control structures. Theorem 3.10 shows that only those flowcharts without Z expressions from the starting node to a node with no successors or which is a persistent set entry node can be written using the above control structures without repeating some computation. Furthermore, theorem 3.5 shows that these flowcharts can be written without the statements IF p THEN R and DO R UNTIL p. This result is an extension of theorem 1 in Peterson, Kasami, and Tokura [PKT73].

An extension to regular expressions which might be useful is to allow a subexpression to be "exitted". In order to do this, the notation would be extended to allow a label to be applied to a subexpression. Then an indication could be made within the labeled subexpression to jump to the point immediately following it.

Example 3.11: The regular expression $(fg)^*(fh+h)$ can be written $R:(f(g+\rightarrow R))^*h$. The subexpression $(f(g+\rightarrow R))^*$ is labeled by R and the notation $\rightarrow R$ means that h is the next symbol to be considered.

This extension doesn't help though in trying to find a notation to express the synchronization for simple serial resources Even a simple expression like that in example 3.11 is not injective and therefore isn't simple serial. Furthermore, theorem 3 of Peterson, Kasami, and Tokura shows that there are still simple serial resources for which the synchronization can't be expressed using a regular expression without repeated names even when this exit notation is allowed.

CHAPTER IV

ELEMENTS

The important property of a simple serial resource is that an operation may only be applied at one state. Thus, only one comparison needs to be made to determine whether or not an operation may execute. Assume, however, that an operation g may execute if the history of executions contains the operation f. Operation g may be applied at many states but most of the information contained in these states is unimportant to g. If the state can be divided into two parts, one of which indicates whether or not f has executed, then g would only need to check that part to determine whether or not it could execute. Furthermore, the part would have only one value at which g could be applied. In an attempt to study this issue, some modification to the notion of state will be made.

For each resource, a new class of object which has a finite number of distinct members will be introduced. Each state, instead of being a single entity, will now be a multiset of these objects. A multiset [K69, page 420] is a set in which members may have multiple occurances. The notation U+V will represent the multiset in which each member of the class occurs the number of times it occurs in U plus the number of times it occurs in V. The notation n*U will represent the multiset in which the number of occurances of each member is n times the number of its occurances in U.

Definition: An object which is used in the composition of a state is an ***element.***

These elements are not each confined to a single state but may be included in several of them. The states are distinguished from each other according to which elements they contain. Thus, no two distinct states are exactly the same multiset of elements. Also, since a state is represented by a group of elements, checking the state variable to see if an operation may execute consists of testing to see that one of several collections of elements is included in the current state.

STATE TRANSITIONS

To convert a state p into a state q requires that every element of p which isn't in q must be removed from the resource state and every element of q which isn't in p must be added.

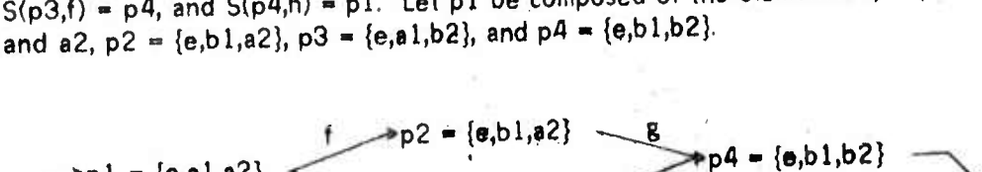
Definition: A *state transition* is the removal of some of the elements from the state of a shared resource followed by the addition of some elements. The notation which will be used for a state transition is <name>:{<elements to be removed>} → {<elements to be added>}. The <name> part is optional and will only be included when necessary.

It may be possible for a state transition to be used at several states. Thus, {e1} → {e2} can transform the state {e1,e3} into {e2,e3} and the state {e1,e4} into {e2,e4}.

An operation on a serial resource will be associated with a collection of state transitions. For each state at which the operation can be applied, one of these transitions will produce the appropriate resulting state. When a process attempts to execute the operation, it will be delayed until all of the elements which are removed by one of these state transitions are present in the current state. These elements are

then removed and at the end of execution the state transition is completed by adding some elements to the state. If more than one process may now continue, a choice must be made. Note that in general an operation doesn't need to remove all of the elements from its starting state but just those which aren't in the resulting state. However, in order that another process doesn't start executing on the resource before this operation finishes, it must not be true that a state transition associated with some operation removes a subset of the remaining elements.

Example 4.1: Consider the regular expression $((fg+gf)h)^*$ and the resulting states p1, p2, p3, and p4 with S(p1,f) = p2, S(p1,g) = p3, S(p2,g) = S(p3,f) = p4, and S(p4,h) = p1. Let p1 be composed of the elements e, a1, and a2, p2 = {e,b1,a2}, p3 = {e,a1,b2}, and p4 = {e,b1,b2}.



If t_f:{e,a1} → {e,b1} is associated with f, t_g:{e,a2} → {e,b2} is associated with g, and t_h:{b1,b2} → {a1,a2} is associated with h, then the proper synchronization results.

Several things should be noticed in this example. First, f and g can each be represented by just one state transition. Second, t_f only removes e and a1 from the current state. When f executes causing a transition from state p1 or p3, a2 or b2 respectively remains part of the current state. Likewise, not all of the elements are removed from the current state when g and h start execution.

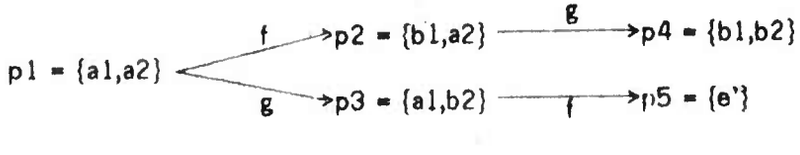
While a state transition doesn't always remove all of the elements of the state at the start of execution of the associated operation, frequently it must remove some

elements which also occur in the resulting state. Of course, it must then add those elements back to the state at the end of execution. In example 4.1, t_f and t_g remove and add e to the state. The reason why e is used in this manner is that otherwise t_f and t_g would remove {a1} and {a2} respectively. Since these are disjoint sets of elements which are both contained in p1, the start of execution of f would leave the elements for which t_g was waiting in the state. Therefore, a process could start executing g before f completed, violating the serial nature of the resource. The solution to this problem is to create a new element which is contained in every state. Then whenever state transitions remove disjoint subsets of a state, they must also remove and add this new element. The state transition t_h doesn't need to remove e since this is the only element remaining during its execution and none of the transitions remove just e.

In addition to assuring that operations execute serially, there is another situation when the state transition from a state p to a state q caused by an operation f must both remove and add the same element. This occurs when the set of elements which must be removed (those which are contained in p but not in q) also form a subset of some state p' different from p. If f can't be applied at p' or if this state transition results in the wrong state when applied at p' then the state transition must additionally wait on some element e which is in p but not in p'.

Consider the following modification to example 4.1.

Example 4.2: Let there be five states with S(p1,f) = p2, S(p1,g) = p3, S(p2,g) = p4, and S(p3,f) = p5. It is irrelevant what operations may be applied at p4 and p5.



The state transition {a2} → {b2} is associated with g and the state transitions {a1,a2} → {b1,a2} and {a1,b2} → {e'} are associated with f.

If {a1} → {b1} was used instead of {a1,a2} → {b1,a2} then f could also take p3 to p4. An element e which occurs in every state isn't needed here since there aren't any state transitions which remove disjoint subsets of a state. Another modification is not to allow f to be applied at p3. The state transition associated with f must still be {a1,a2} → {b1,a2} to prevent it from being applied at p3.

Actually, in an implementation a state transition doesn't need to remove and then add an element only to prevent being used at a state where it shouldn't be. A check of the state to make sure that the element is present is all that is needed. However, removing the element is acceptable and is consistent with the model of synchronization as presented, so no further extension will be given for this special case.

SUBSTATES

If a state transition may occur more than once consecutively from a state, then each element which it removes and doesn't return must have more than one instance in the original state.

Example 4.3: Consider the regular expression $(fgg)^*$. This may be represented

with states p1 = {e1,e1,e2}, p2 = {e2,e2,e2}, and p3 = {e1,e2,e2} with the state transitions t_f:{e1,e1} → {e2,e2} and t_g:{e2,e2} → {e1,e2} corresponding to f and g respectively.

Since t_g removes e2 and adds e1, e2 must occur at least twice in p2 and e1 must occur at least twice in p1.

Definition: The ***multiplicity*** of an element e in the state p of a shared resource is the number of instances of e in p.

In example 4.3, e2 has a multiplicity of three in state p2, two in state p3, and one in state p1. Since t_g removes e2 twice, e2 must have a multiplicity of at least two in the current state in order that t_g may be used. Since this is not the case in p1 and since t_g is the only state transition associated with g, any process which tries to execute g when the state is p1 will block.

It is now necessary to return to the situation where the elements which a state transition must remove from a state p form a subset of some state p' at which the associated operation can't be applied. Such is the case in example 4.3 where t_g must remove {e2} from p2 and from p3 but e2 is also in p1, a state at which g can't be applied. In this example, however, p1 also contains the only other element, e1. Therefore, t_g can't remove an element which is contained in p2 and p3 but not in p1. Only the multiplicities are different. Thus, to prevent g from executing at p1, some element must be removed in a greater amount than its multiplicity in p1. Here that is possible since e2 has a greater multiplicity in p2 and p3 than it does in p1.

It was stated above that e2 must have a multiplicity of at least two in p2. In fact, it has a multiplicity of three. Also, t_g removes e2 twice rather than once and then

adds the second one back. This is only partly because e2 has a non-zero multiplicity in p1. The reason why e2 has a multiplicity in each state of one greater than it needs to be is that it is used to perform the same function that e does in example 4.1. Here, two processes could execute g simultaneously from state p2 otherwise. In general, if a state transition can be used n times in sequence from a state p, then it can be prevented from being used twice simultaneously by removing some element n times and adding it n-1 times. The multiplicity of this element should be 2n-1 in p. Thus, after n applications of the state t ansition, the multiplicity of this element in the current state is n-1 and it can't be applied again.

It might be true that there are states p and q such that not only is every element in p also in q but the multiplicity of each of these elements is at least as great in q as it is in p.

Definition: If p and q are states, then p is a ***substate*** of q, denoted $p \subset q$, if ($\forall$e, e an element) the multiplicity of e in p isn't greater than the multiplicity of e in q and ($\exists$e', e' an element) the multiplicity of e' in p is less than the multiplicity of e' in q.

If p is a substate of q then it is clear that any state transition, and therefore any operation, which can be used at p can also be used at q. Furthermore, the state resulting from using such a transition at p must be a substate of the state resulting from using it at q. This is true since the elements not removed from p are a subset of those not removed from q.

It is possible to extend the concept of an arc progression to state transitions. For every arc progression $(p_0,f_1)...(p_{n-1},f_n)$ there is a corresponding string of state transitions $t_1...t_n$. Each t_i is the state transition caused by executing f_i from state p_{i-1}.

Composing these state transitions then yields a state transition which corresponds to executing the entire arc progression.

Definition: A ***composed state transition*** is the result of using a string of state transitions, corresponding to some arc progression, in sequence.

A composed state transition t corresponding to $t_1 \ldots t_n$ may be created in the following manner. If an element is added by t_i and removed by t_j where $i<j$, then this addition and removal cancel each other. After all possible cancellations are made, t removes the sum of the elements which the t_i's remove and adds the sum of the elements which the t_i's add.

If the set of elements which a state transition removes is a subset of those that it adds, then any state that contains the elements for which the transition waits is a substate of the resulting state. Thus, the state transition could then be applied again. Such a state transition can therefore be used an arbitrary number of times in succession. Extending this observation to composed state transitions produces the following results.

Theorem 4.4: If a shared resource R has a finite number of elements, then the number of states is finite iff there are no states p and q of R such that $p \subset q$ and there is a composed state transition $t: p \rightarrow q$.

Proof: If there are 2 such states, then an infinite number of states may be generated by repeated use of t. On the other hand, if the number of states is infinite, then the multiplicity of some element e must be unbounded. Hence, there must be states $p_1, \ldots, p_i, \ldots$ such that $(\forall i, i \geq 1)$ e has a greater multiplicity in p_{i+1} than in p_i and there is a composed state

transition t_i:$p_i \rightarrow p_{i+1}$. If $p_1 \subset p_i$ for some i, then the theorem is proved. Otherwise, each p_i has at least one element with lower multiplicity than in p_1. Since there are finitely many elements and infinitely many p_i's, for some element e1 there are infinitely many of the p_i's which have a lower multiplicity of e1 than does p_1. If the multiplicity of e1 in p_1 is k, then these may be divided into k classes representing each value of the multiplicity of e1 less than k. One of the classes must have an infinite number of members p_1',...,p_i',... such that the multiplicity of e1 is the same for each p_i' and ($\forall i, i \geq 1$) e has a greater multiplicity in p_{i+1}' than in p_i' and there is a composed state transition t_i':$p_i' \rightarrow p_{i+1}'$. The above procedure may then be repeated. It must terminite since there are only a finite number of elements.

Corollary 4.5: If there are a finite number of states and if p and q are states such that $p \subset q$ and there is an arc progression from q to p, then q isn't in any persistent set.

Proof: There can be no arc progression from p to q.

IMPLEMENTATION

For most serial resources, the implementation based on elements will be more complex than that based on states. However, it will be seen that this isn't true for a special class of these resources. Before this class is presented, though, a general implementation will be introduced.

In the previous implementation, the state was represented either by a single

variable or by a set of boolean semaphores. With elements, the state must be represented by a set of variables. Each of these variables is used to keep track of the current multiplicity of one of the elements. Likewise, for each state transition associated with an operation, the amount of each element that it removes and adds must be stored. There are two ways to do this. The first is to keep the amounts for each of the elements, including a zero for those that it doesn't remove or doesn't add. The other way is to save only the nonzero amounts and to label each with the element to which it corresponds. Since these labels require space, the second method will use more storage unless most of the state transitions are sparse in that they remove only a small percentage of the elements.

When a process attempts to execute an operation, each of the various state transitions associated with the operation must be compared with the state. This is basically the same procedure that was used in the implementation described in chapter II. The number of state transitions involved may be fewer than the number of states, but each comparison now requires checking the multiplicity of each of the elements which must be removed. Thus, several variables must be compared rather than just one. The number of comparisons which will be made in the worst case, when the process becomes blocked, will be the sum of the number of elements which must be removed by each of the state transitions associated with the operation. In addition, if the first method above is used to store the state transitions, for each state transition tried, every element's value must first be compared with zero. When a match is found, the identity of the appropriate state transition must be saved so that the proper one will finish when the operation completes its execution.

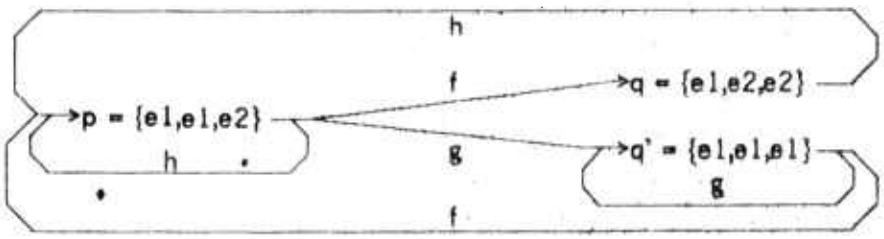
If none of the state transitions can proceed, then the process must be put on a waiting list. The waiting lists should be organized as before. Either each set of states at which an operation may be applied has a waiting list or else there is a single list.

When an operation completes execution, the state transition resumes by adding elements to the current state. Instead of being unique, the resulting state will be one of several depending on which elements the state variable already contained. If there is a single waiting list, each process is checked in turn by comparing the current state with the elements removed by each of the state transitions associated with the operation the process is attempting to execute. If there are several waiting lists, then they are ordered according to the length of time that the top element has been waiting or some other priority scheme. Using this ordering, the top process on each list is checked as in the one list case. When a process is found which can continue, the multiplicities of the appropriate elements are decremented in the current state.

The several waiting list implementation may now seem to be the same as the one using a single list. The difference is that with the several list scheme, if the top process of a list fails, none of the other processes on that list will be tested to see if it can continue. Thus, if execution of an operation f is enabled, with a single list several processes attempting to execute another operation g might be higher on the list than the first process attempting to execute an f. Each of these processes will be tested while with several waiting lists only one such process would be tested. In addition, a further simplification can be made when several waiting lists are used. Usually, only a few of the states are possible results from completing a state transition. Some of the operations won't be able to begin execution at any of these

states. Therefore, the waiting lists of processes trying to execute these operations need never be checked.

Returning to example 2.11, let p = {e1,e1,e2}, q = {e1,e2,e2}, and q' = {e1,e1,e1} and let t_f:{e1,e1} → {e1,e2} be associated with f, {e1,e1,e2} → {e1,e1,e1} and {e1,e1,e1} → {e1,e1,e1} be associated with g, and {e1,e2,e2} → {e1,e1,e2} and {e1,e1,e2} → {e1,e1,e2} be associated with h.



As explained before, since f may go twice in a row from q' and it removes e1, to prevent two processes from executing f in parallel e1 should be removed twice and added once. It must also have a multiplicity of three in q. When a process tries to execute f, it must wait until the variable for e1 has a value of at least two. A process trying to execute g must wait until the variable associated with e1 has a value of three or else until e1 has a value of two and e2 has a value of one. There will be two waiting lists as before. One is for processes trying to execute either an f or a g and the other for those trying to execute an h. When t_f completes, the resulting state will either be p or q. Processes trying to execute either an f or a g will only be allowed to proceed if the state is p. Thus, before the list for processes waiting to either execute an f or execute a g can be searched, the identity of the current state must be determined. An h can be applied at either of these states, so the list for processes trying to execute it must be checked. When an h finishes executing, the state must be

p and both lists will be examined for waiting processes. Likewise, when a g finishes executing, the state must be q' and only the list for f and g will be checked.

ASSIGNING ELEMENTS TO STATES

The synchronization as studied so far is expressed in terms of states or else using a notation, such as regular expressions, which can be converted into states. In order for elements and an implementation based on them to be useful, it must be possible to convert from states into multisets of elements.

For a resource with states $p_1,...,p_n$, one way to assign multisets of elements to these states is to create n pairs of elements. For each pair a_i and b_i where $1 \le i \le n$, include a_i in state p_i and b_i in each state p_j for $i \neq j$. Thus, each state contains n elements each with a multiplicity of one. A state transition from state p_i to state p_j can be written as $\{a_i,b_j\} \rightarrow \{b_i,a_j\}$. The elements b_k for $k \neq i$ and $k \neq j$ are in both p_i and p_j and therefore don't have to be included in the transition. This transition may only be used at p_i since that is the only state containing a_i. In example 4.1, p1 = {a1,b2,b3,b4}, p2 = {b1,a2,b3,b4}, p3 = {b1,b2,a3,b4}, and p4 = {b1,b2,b3,a4}. The state transitions {a1,b2} → {b1,a2} and {a3,b4} → {b3,a4} are associated with f, {a1,b3} → {b1,a3} and {a2,b4} → {b2,a4} are associated with g, and {b1,a4} → {a1,b4} is associated with h.

This assignment of elements to states leads to the worst case in that the maximum number of state transitions will be needed. In order to reduce the number of state transitions associated with an operation, some of the elements in the states at

which the operation can be applied and in the resulting states must be replaced by other elements. To do this, two such transitions are set equal. Thus, if {e}+U → {e'}+U' and V → V' are both associated with an operation, they are set equal and the equation is solved. This is done by letting e = V+U' and e' = V'+U. This substitution is made in every state transition and also in each state. The first state transition then becomes V+U'+U → V'+U+U' which reduces to the second. In example 4.1, setting {a1,b2} → {b1,a2} = {a3,b4} → {b3,a4} yields a1 = {a2,a3,b4} and b1 = {b2,b3,a4}. The states are now p1 = {a2,b2,a3,b3,b4,b4}, p2 = {a2,b2,b3,b3,a4,b4}, p3 = {b2,b2,a3,b3,a4,b4}, and p4 = {b2,b2,b3,b3,a4,a4}. Operation f is now only associated with the state transition {a3,b4} → {b3,a4}, g is associated with {a2,b4} → {b2,a4} and {a2,a3,b3,b4} → {b2,a3,b3,a4} which reduces to {a2,b4} → {b2,a4}, and h is associated with {b2,b3,a4,a4} → {a2,a3,b4,b4}.

Several things must be noted about the above algorithm. First, the multiplicity of some of the elements may be greater than one in some of the states. In the example, b2 has a multiplicity of two in p3 and in p4. It is therefore possible that some elements might be removed or added more than once by a state transition. In the transition associated with h, a4 is removed twice and b4 is added twice. If such a transition is set equal to another and an element a_i which is removed or added n times by the state transition is solved for, the result will be of the form $n*a_i = U$ and $n*b_i = U'$ where U and U' are multisets of elements. But a_i or b_i might have a multiplicity which isn't a multiple of n in some state. Simple substitution would therefore result in fractions of elements. This problem can be corrected by multiplying the multiplicity of every element in every state by n. The solution to the equality of the state transitions will then be $n^2*a_i = n*U$ and $n^2*b_i = n*U'$ which reduces to $n*a_i = U$ and

$n*b_i = U'$. Now, however, the multiplicities of a_i and b_i in every state must be multiples of n.

Next, some of the elements may have a multiplicity of at least one in every state. This is true of b2 and b3 above. Subtracting the minimum such multiplicity from every state won't change any of the state transitions. The result is that the states can be simplified. In example 4.1, the states become p1 = {a2,a3,b4,b4}, p2 = {a2,b3,a4,b4}, p3 = {b2,a3,a4,b4}, and p4 = {b2,b3,a4,a4}. Third, both elements of a pair may now be in a state. Thus, a4 and b4 are both in p2 and in p3.

Finally, a state transition $U \rightarrow V$ for some multisets of elements U and V can always be written as $k*\{a_i\}+U' \rightarrow k*\{b_i\}+V'$ where U' and V' are also multisets and which contain neither a_i nor b_i. This can be shown by assuming that the sum of the multiplicities of the two elements in any pair is the same in every state. This is certainly true for the initial assignment where this sum has the value one for each pair. Thus, if a_i is removed k times from a state then b_i must be added k times. Assume that e has multiplicity m and b_i has multiplicity n in state p and e has multiplicity m' and b_i has multiplicity n' in state q. If the sums are the same in every state then m+n = m'+n'. If m>m', then in the state transition from p to q, a_i must be removed m-m' times and b_i must be added n'-n = m-m' times. If solving for a_0 and b_0 produces $a_0 = U''$ and $b_0 = V''$ and $a_i \in U''$, then the multiplicity of a_i in U'' must equal the multiplicity of b_i in V''. When a_0 and b_0 are substituted for in each state, since the sum of their multiplicities are the same, the sum of the multiplicity of a_i and b_i must be the same in every state.

It isn't always possible to set two state transitions equal. If it was, then the

same transition could be used for every operation of a resource just by setting all of the state transitions equal to each other. There are three situations for which state transitions can't be set equal. The first occurs when state transitions $U \rightarrow U'$ and $V \rightarrow V'$ are set equal and an element a_i is solved for which is in both U and V' with multiplicities m and n respectively. It must also be true that b_i is in both U' and V. The result must be that $m*a_i = n*\{b_i\}+U''$ and $m*b_i = n*\{a_i\}+V''$ for some multisets U" and V". Since these solutions are mutually recursive, no such element must ever be solved for. If every element of U is also in V' and every element of V' is in U, then no element can be solved for and the two state transitions can't be set equal.

Another situation occurs when substituting multisets of elements V and V' for elements a_i and b_i respectively causes two different states to become equal. Such a substitution can't be allowed. A check for this situation can be made as follows. If the multiplicity of a_i in a state p minus the multiplicity of a_i in a state q is some number n, then p and q will become equal if $p-n*\{a_i\}+n*V = q-n*\{b_i\}+n*V'$. If n=0 ($a_i$ has the same multiplicity in p and q) then this check is unnecessary. If there are no elements a_i and b_i from two state transitions which when substituted for don't collapse some states into one, then these transitions can't be set equal.

A final situation occurs when making a substitution causes the intersection of the states at which some operation can be applied to become contained in another state. If such a substitution were allowed, then there would be no element that a state transition could remove and that was in every state at which the operation could be applied but not in the other state. Therefore, the operation couldn't be associated with just one state transition. A check must be made that this condition doesn't hold

after the substitution for any operation which can be applied at more than one state. If for two state transitions every element which can be solved for causes this condition, then the transitions can't be set equal

After all possible substitutions have been made, it may be possible to reduce the number of elements in each state. If the multiplicity of some element e' is at least as great as the multiplicity of an element e in every state, then create a new element e" = {e,e'}. A substitution is made by subtracting the multiplicity of e from that of e' in every state, letting e" have the same multiplicity as e, and deleting e. A substitution must be made in the state transitions also. If e is removed (added) then e" must be removed (added) instead and e' must be added (removed). If e' is now both added and removed, these can cancel as before. This procedure can never cause two states to collapse into one, but it might cause the intersection of the states at which a transition can be used to become contained in another. Therefore, a check for this situation must be made before a substitution can be allowed.

Returning to example 4.1, every state containing a3 also contains b4. Therefore, let a1' = {a3,b4}. The states become p1 = {a1',a2,b4}, p2 = {a2,b3,a4,b4}, and p3 = {a1',b2,a4}, the state transition associated with f becomes {a1'} → {b3,a4}, and the state transition associated with h becomes {b2,b3,a4,a4} → {a1',a2,b4}. The state transition associated with g remains {a2,b4} → {b2,a4} and p4 still equals {b2,b3,a4,a4}. Now every state containing b2 also contains a4 so letting b2' = {b2,a4} produces p3 = {a1',b2'}, p4 = {b3,a4,b2'}, the state transition {a2,b4} → {b2'} to be associated with g, and the state transition {b3,a4,b2'} → {a1',a2,b4} to be associated with h. It is now possible to let {a2,b4} = a2' and {b3,a4} = b1'. The result is that p1 = {a1',a2'}, p2 =

{b1',a2'}, p3 = {a1',b2'}, and p4 = {b1',b2'}. The state transition associated with f is {a1'} → {b1'}, {a2'} → {b2'} is associated with g, and {b1',b2'} → {a1',a2'} is associated with h.

After the synchronization relationships have been reduced as much as possible, a check must be made to be sure that the set of elements that each state transition removes isn't contained in some state at which the transition shouldn't be used. If it is, an element from the intersection of the states at which the state transition can be used but which isn't already removed should be both removed and added by the state transition. This process should continue until the elements which it removes are no longer contained in any states at which the transition shouldn't be used. When including these elements in the state transition, for reasons that will become clear later, any which have a multiplicity bound by one should be included first. Also, if any two state transitions remove disjoint subsets of a state but their associated operations should execute serially, a new element should be added to every state and these two transitions must both remove and then add this element. Thus, in example 4.1, a new element e must be added to every state which the state transitions associated with f and g each removes.

SINGLE TRANSITION OPERATIONS

As can be seen from examples 4.1 and 4.3, often one state transition can be used to represent the state change caused by applying an operation at any one of several states. Thus, in example 4.1 the state transition {e,a1} → {e,b1} can be used to change p1 into p2 and p3 into p4. Likewise, in example 4.3, {e2,e2} → {e1,e2} can be used to change p2 into p3 and p3 into p1.

Definition: An operation is ***single transition*** if one state transition can be used to represent exactly those state changes which the operation can cause. A resource is ***single transition*** if every operation defined on it is single transition.

The advantage of a single transition operation is that only one transition needs to be checked at the start of the operation. Also, the identity of this transition doesn't need to be saved during the execution of the operation.

Trivially, every operation which is both injective and projective is single transition. Thus, a simple serial resource is single transition. For other resources, though, it may not be possible to make every operation single transition. The following result shows that every single transition operation must be injective.

Theorem 4.6: It isn't possible for a state transition to take different states p and p' into the same state q.

Proof: Assume that there are states p, p', and q such that some state transition t:V → V' takes p and p' into q. Since t can be used at p and p', there must be multisets U and U' such that p = U+V and p' = U'+V. Using t at p results in q = U+V' and using it at p' results in q = U'+V'. Therefore, U = U' and p = U+V = p' and p and p' aren't different states.

If state transitions U → U' from a state p to a state q and V → V' from p' to q are set equal using the algorithm above, the result will be that p and p' become equal. This may be seen by solving for some element e with multiplicity n in U. The result is that n*e = U'-n*{e'}+V and n*e' = U-n*{e}+V' which becomes n*e = U+V'-n*{e'}. Subtracting the two solutions for e yields U-U' = V-V'. But p = q-U'+U = q-V'+V = p'. This result can also be extended to composed state transitions. Thus, if S(p,f) = p', S(p',g) = p", S(q,g) = q', and S(q',f) = p", then either f or g isn't single transition.

The next result shows that if a group of single transition operations execute from a state, the resulting state will always be the same regardless of the ordering. This is a commutative law for single transition operations

Theorem 4.7: If a state transition t takes state p into p' and state q into q' and a state transition t' takes p into q and p' into p", then q' = p".

Proof: Assume that t = U → U' and t' = V → V'. Then p' = p-U+U' and p" = p'-V+V' = p-U+U'-V+V'. Also, q = p-V+V' and q' = q-U+U' = p-V+V'-U+U' = p".

Thus, for the synchronization expressed by (f g h+g f i)* either f or g can't be single transition since f g and g f executing from the initial state result in different states.

The third result shows that if an operation is single transition and it can be applied n times in a row starting at a state p with the result being state p for some n≥1, then the result of applying it at any state q (including p) must be q.

Theorem 4.8: If a state transition U → V can be used n times in a row starting at a state p with the result being p for some n≥1, then U = V.

Proof: After using the state transition n times from p, the state will be p-n*U+n*V = p. Therefore, n*U = n*V.

Thus, if the synchronization for a serial resource is given by the regular expression (f+(g g))* then operation g can't be single transition. If for some m, an operation can be applied m times at a state p with the result being state q using a state transition U → U' and it can be applied at q with the result being p using a state transition V → V', then V' = m*U and therefore U and V' contain the same elements and it won't

be possible to set them equal. This theorem can also be extended to strings of operations. Combining it with the commutative law shows that if executing an f from a state p followed by executing a g results in state p and if executing a g from some state q followed by executing an f results in a state q', then either f or g isn't single transition.

The final result shows that if a single transition operation f can execute several times in a row from a state p with the result being state q and another single transition operation can be applied at both p and q then it can also be applied at any of the intermediate states in the string of f's.

Theorem 4.9: If there are states $p_0,...,p_n$ and a state transition $t:U \rightarrow V$ such that $(\forall i,\ 1 \leq i \leq n)$ t takes p_{i-1} into p_i and there is a state transition t' which removes the multiset of elements U' and can be used at p_0 and p_n, then $(\forall i,\ 0 \leq i \leq n)$ t' can be used at p_i.

Proof: It must be true that $(\forall i,\ 0 \leq i \leq n)$ $p_i = p-iU+iV$. Since t' can be used at p_0 and p_n, for every element e the multiplicity of e in U' can't be greater than the multiplicity of e in either p or in $p_n = p-nU+nV$. Let m_e be the difference between the multiplicity of e in V and the multiplicity of e in U. Thus, the multiplicity of e in p_i must be the multiplicity of e in p plus $i*m_e$. If $m_e \geq 0$, then the multiplicity of e in p_i must be at least as great as the multiplicity of e in p which is at least as great as the multiplicity of p in U'. If $m_e < 0$ then $i*m_e \geq n*m_e$ and therefore the multiplicity of e in p_i is at least as great as the multiplicity of e in p_n which is at least as great as the multiplicity of e in U. Therefore, t' can be used at p_i.

This theorem shows that for the synchronization expressed by the regular expression $(g+ffg^*h)^*$ either f or g can't be single transition. If the above algorithm was applied, the intersection of the states at which g can be applied would be contained in each of the states between the two f's.

Two restrictions to a single transition resource are allowing a transition to remove at most one occurrence of each element and allowing a transition to remove only one element but by any amount. These restrictions are equivalent to the resources which can be implemented using P-V multiple and P-V chunk respectively and placing bounds on the semaphores. Since any synchronization which can be expressed using P and V can also be expressed using P-V multiple, the resources which can be synchronized with P and V and bounded semaphores form a subclass of the single transition resources.

BOOLEAN ELEMENT RESOURCES

Single transition operations need only attempt one state transition in order to execute and therefore the same set of elements is always added to the state upon completion. However, several variables must still be checked when a process tries to execute such an operation and also whenever an attempt is made to remove it from a waiting list. For a subclass of the single transition operations, though, the implementation can be changed so that only one variable must be checked to determine if the operation may start execution.

Definition: A state transition is ***boolean element*** if every element which it removes has a multiplicity of at most one in any state. A shared resource is ***boolean element*** if it is single transition and every element has a multiplicity bounded by one.

Thus, every state transition associated with an operation of a boolean element resource must be boolean element. The resource in example 4.1 may easily be seen to be boolean element.

If a state transition is boolean element, then it is always possible to alter the implementation by adding new elements such that the multiplicity of only one element needs to be checked. Assume that the state transition removes n elements. Create a new element e such that at any time its multiplicity is the sum of the multiplicities of these n elements. Thus, whenever one of these elements is added to the state, the multiplicity of e is increased by one and whenever one is removed, the multiplicity of e is decreased. Since the multiplicity of each of these elements is bound by one, their sum, and therefore the multiplicity of e, is bound by n. Also, the multiplicity of e will reach n exactly when all of these elements are part of the state. The state transition now only must wait until the multiplicity of e equals n. At such a time, the rest of the elements which it must remove are guaranteed to be part of the state in the appropriate multiplicity. A simplification can be made by deleting any element which no state transition waits on.

Returning to example 4.1, let e1 = {e,a1}, e2 = {e,a2}, and e3 = {b1,b2}. The state transition {e,a1,e1,e1,e2} → {e,b1,e1,e2,e3} only needs to wait for the multiplicity of e1 to be two and corresponds to f, {e,a2,e2,e2} → {e,b2,e1,e2,e3} only needs to wait for the multiplicity of e2 to be two and corresponds to g, and {b1,b2,e3,e3} → {a1,a2,e1,e2} only needs to wait for the multiplicity of e3 to be two and corresponds to h. Since no state transition waits for e, a1, a2, b1, or b2, these elements may be deleted. The result is that p1 = {e1,e1,e2,e2}, p2 = {e1,e2,e2,e3},

p3 = {e1,e1,e2,e3}, and p4 = {e1,e2,e3,e3}. The operation f corresponds to the state transition {e1,e1,e2} → {e1,e2,e3} but it doesn't need to check e2, g corresponds to {e1,e2,e2} → {e1,e2,e3} but doesn't need to check e1, and h corresponds to {e3,e3} → {e1,e2}.

An alternate simplification can also be made to the implementation of a boolean element resource. The state can be represented with a string of bits. Each zero bit means that the corresponding element is present and a one means that it isn't. To check for a group of elements a mask is used. Every one in the mask indicates an element which is needed. If the result of performing an AND operation between the mask and the state is zero, then the state transition has succeeded. To remove the appropriate elements from the state, the bit string is ORed with the mask. To add elements to the state, another mask with a zero for each element being added and a one for the rest of the elements is used. This mask is ANDed to the current state bit string.

Since the implementation of a boolean element resource involves a fairly small amount of overhead, it would be reasonable to restrict a programming system to such resources. To help make such a restriction, a notation which corresponds to this class of synchronization is desirable.

Definition: A ***multiple regular expression*** is a set of regular expressions. It is ***restricted*** if every member of the set is restricted.

A multiple regular expression is interpreted such that the synchronization expressed by each of the member expressions must be satisfied.

Example 4.10: The restricted multiple regular expression $\{(f(g+h))^*,(g\ h)^*\}$

means that execution of f must alternate with the execution of g or h and that execution of g and h must alternate. This is the same synchronization as that expressed by the regular expression $(f\ g\ f\ h)^*$.

It will now be shown that the restricted multiple regular expressions correspond exactly to the boolean element resources.

Theorem 4.11: A resource is boolean element iff the synchronization on it can be expressed with a restricted multiple regular expression.

Proof: If a resource is boolean element, then for every element e create a new element e' and include e' in every state which doesn't contain e. The state transitions must be changed so that if e is removed but not added then e' must be added and if e is added but not removed, then e' must be removed. A restricted regular expression will be created for every pair of elements e and e'. Assume that operations $f_1,...,f_i$ remove e and add e', operations $f_{i+1},...,f_j$ remove e' and add e, operations $f_{j+1},...,f_k$ remove and add e, and the rest of the operations neither remove nor add e and e'. If e is in the initial state, then the regular expression can be written as $(f_{j+1}+...+f_k+((f_1+...+f_i)(f_{i+1}+...f_j)))^*$ and if e' is in the initial state, then the regular expression can be written as $((f_{i+1}+...+f_j)(f_{j+1}+...+f_k)^*(f_1+...+f_i))^*$. If a resource can be expressed as a restricted multiple regular expression, then it forms a set of simple serial resources. Assume that the states of each of these resources are disjoint and use them as the elements of the complete synchronization relationships. The initial state is composed of the elements representing the initial states of the various simple serial resources. Since each of

these resources can only be in one state at a time, the multiplicity of each element is bound by one. Each operation removes the elements corresponding to the states at which it could be applied and adds the elements corresponding to the states which could result from its execution in the various simple serial resources. Since there is only one state at which it can be applied in each such resource, it must be single transition.

The restricted multiple regular expression $\{(f\ h)^*,(g\ h)^*,(f+g)^*\}$ can be used to express the synchronization of the resource in example 4.1. The expression in example 4.10 corresponds to states p1 = {a1,a2}, p2 = {b1,a2}, p3 = {a1,b2}, and p4 = {b1,b2} and state transitions {a1} → {b1} associated with f, {b1,a2} → {a1,b2} associated with g, and {b1,b2} → {a1,a2} associated with h.

While restricted multiple regular expressions can be used to express the synchronizations for the boolean element resources, trying to understand several expressions simultaneously is harder than understanding a single expression. In particular, it is easier to include deadlock situations. An example is $\{(f\ g)^*,(g\ f)^*\}$. No process will ever be allowed to execute either an f or a g. In order to help prevent such situations from occurring, a compiler for a language which allows synchronization to be expressed using multiple regular expressions would need to create the states and successor function. States at which no operation can be applied and the auxiliaries of each persistent set then can be found. If there is no state in any of the subexpressions at which no operation can be applied but there is one for the resulting synchronization relationships, then a warning should be given. Likewise, if for every expression that some operation is in it is an auxiliary of every persistent set, then it should be in every persistent set of the result.

CHAPTER V

CONCURRENT RESOURCES

When several processes can operate on a shared resource in parallel, usually each process may be considered to be operating on a different part of the resource, each with its own set of operations and synchronization relationships. For example, consider a ring of buffers which several processes may access simultaneously. Each buffer in the ring may be thought of as a unique resource which may only be accessed by one process at a time with the operations insert and remove alternating. However, sometimes it isn't possible to consider a resource which can be operated on in parallel as being composed of several independent parts.

Example 5.1: While a disk transfer is occurring, the process which controls the disk can be selecting the next transfer. The new request may not be passed to the disk, though, until both the disk has finished its transfer and the selection is completed.

In actual practice, the disk transfer resource will be more complex. A delay operation which is part of a clock resource will be used to insure that a selection isn't made until the transfer has almost completed. The selection operation first calls this operation before it makes the selection.

Another example occurs when several processes are allowed to read or copy a file simultaneously. However, reading and copying are not allowed while the file is being written.

Definition: A *concurrent resource* is a shared resource on which it is possible for more than one process to operate at a time.

The final synchronization to be studied is that of concurrent resources.

PROLOGUES AND EPILOGUES

So far, an operation has been viewed as a group of state transitions only one of which is used each time it is executed. This was acceptable since the resource state couldn't be changed during the execution of the operation by another process starting or completing execution. Therefore, the execution of an operation could be viewed as being instantaneous. When processes can operate on a resource in parallel, though, this is no longer true. In this case, the start and end of an operation must be treated as separate state transitions.

It is possible to handle concurrent resources within the model developed for serial resources by introducing for each operation which must be synchronized a pair of null operations which have no effect on the resource. One of the null operations will be called before execution of the operation and the other will be called after execution. The synchronization is then expressed in terms of the null operations which must be used serially.

Definition: The *prologue* of an operation f defined on a concurrent resource is a null operation which must be called by f at the start of its execution. The *epilogue* of f is a null operation which must be called by f at the end of its execution. A *perilogue* is either a prologue or an epilogue.

Since the perilogues must be used serially, corollary 2.5 shows that the synchronization for a finite state concurrent resource can be expressed as a regular expression of the perilogues.

A process must wait to execute an operation until its prologue can be applied at the current state of the resource. The state change associated with the prologue is then made without entering a null state. This can be done since the prologue has no code and may be thought of as executing instantaneously. If elements are used, this means that a state transition removes and adds the appropriate elements simultaneously without entering some intermediate state. When the operation finishes execution, some state change corresponding to the epilogue must be made. Once again, this state change can be made instantaneously.

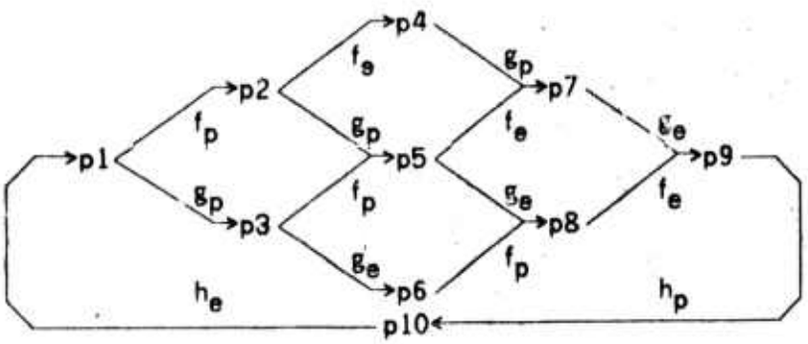
While the prologue of an operation may block until the resource enters a state at which it may be applied, it should always be the case that an epilogue will be able to be applied immediately upon completion of the corresponding operation. When the epilogue is attempted, the operation has already made all of its accesses to the resource and reliability can't be improved by a delay at this point. Therefore, the epilogue must be able to be applied at every state which can result from the prologue in case no other operation starts or stops during execution of the operation. In general, if an epilogue can be applied at a state p and some other perilogue can also be applied at p with the result being state q, then the epilogue must be able to be applied at q.

In the implementation of concurrent resources, the waiting lists must be checked more often than they were in the implementation of serial resources. When a process is allowed to execute an operation, it causes a state change to take place. Therefore, some of the processes which are blocked and on a waiting list may now be able to execute. A check of the waiting processes must be made. This procedure continues

untll none can go. Thus, the waiting lists must be checked whenever an operation starts and whenever it finishes, twice as often as for a serial resource.

Using prologues and epilogues, concurrent resources may be implemented using the method described in chapter II based on the successor function. However, even simple resources will have a complicated implemantation. On the other hand, some of these resources will turn out to be boolean element and can be implemented simply using the method described in chapter IV.

Example 5.2: Consider a modification to example 4.1 which allows operations f and g to be executed in parallel. There are now ten states with $S(p1,f_p) = p2$, $S(p1,g_p) = p3$, $S(p2,f_e) = p4$, $S(p2,g_p) = S(p3,f_p) = p5$, $S(p3,g_e) = p6$, $S(p4,g_p) = S(p5,f_e) = p7$, $S(p5,g_e) = S(p6,f_p) = p8$, $S(p7,g_e) = S(p8,f_e) = p9$, $S(p9,h_p) = p10$, and $S(p10,h_e) = p1$.



The notation f_p and f_e is used to respectively indicate the prologue and epilogue of operation f. Using elements, the states become:

p1 = {a1,a2}	p2 = {c1,a2}	p4 = {b1,a2}
p3 = {a1,c2}	p5 = {c1,c2}	p7 = {b1,c2}
p6 = {a1,b2}	p8 = {c1,b2}	p9 = {b1,b2}
	p10 = {e}	

and the prologues and epilogues become:

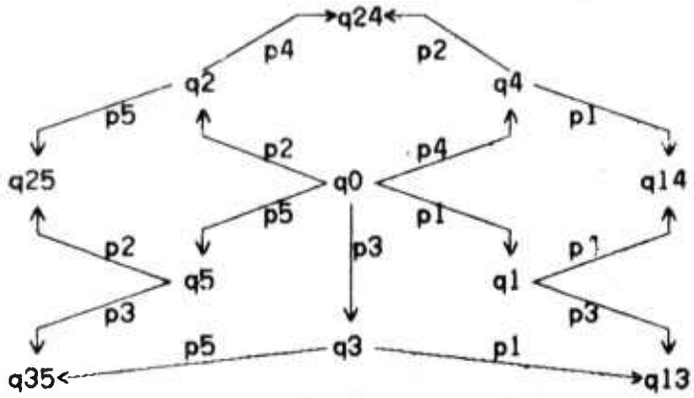
f_p: {a1} → {c1}	g_p: {a2} → {c2}	h_p: {b1,b2} → {e}
f_e: {c1} → {b1}	g_e: {c2} → {b2}	h_e: {e} → {a1,a2}

Each of the perilogues f_p, f_e, g_p, and g_e may be applied at three states and none is projective. Thus, when an operation is called, three states must be compared with the initial state. When it finishes, this comparison must be done again to determine the resulting state. On the other hand, the resource is boolean element and each perilogue is only associated with one state transition.

This example is essentially the same as example 5.1. The operation to select the next disk transfer corresponds to f, the disk transfer itself corresponds to g, and the issuing of the transfer command corresponds to h. The initial state for this example must be p6 which allows a command to be selected but requires that the command be passed to the disk before a transfer starts.

Another example of a synchronization problem involving a concurrent resource which is boolean element but is complicated when described using states and the successor function is the famous "Five Dining Philosophers" problem [D68].

Example 5.3: The states of the "Five Dining Philosopher" problem are q0 which corresponds to no philosopher eating, q1, q2, q3, q4, and q5 corresponding respectively to just p1 eating, just p2 eating, just p3 eating, just p4 eating, and just p5 eating, and q13, q14, q24, q25, and q35 corresponding respectively to p1 and p3 eating, p1 and p4 eating, p2 and p4 eating, p2 and p5 eating, and p3 and p5 eating. In the following diagram, going along an arc in the direction of the arrow is the prologue of the operation and going in the opposite direction is the epilogue of the operation.



This resource can be shown to be shown to be boolean element by assigning elements to states as follows:

q0 = {e0,e1,e2,e3,e4}	q1 = {e5,e2,e3,e4}	q5 = {e1,e2,e3,e9}
q3 = {e0,e1,e7,e4}	q13 = {e5,e7,e4}	q35 = {e1,e7,e9}
q4 = {e0,e1,e2,e8}	q14 = {e5,e2,e8}	q25 = {e6,e3,e9}
q2 = {e0,e6,e3,e4}	q24 = {e0,e6,e8}	

and by using the following prologues and epilogues:

$p1_p$: {e0,e1} → {e5}	$p2_p$: {e1,e2} → {e6}	$p3_p$: {e2,e3} → {e7}
$p4_p$: {e3,e4} → {e8}	$p5_p$: {e0,e4} → {e9}	

Each epilogue f_e is the reverse of the prologue f_p.

Once again, none of the perilogues is projective, but a simple implementation is possible based on the elements.

For shared resources, a process might call any of the operations at any time. Thus, the resource can be in any of its states when an attempt is made to use an operation if the resource is serial or to use a prologue if the resource is concurrent. However, there are some states at which epilogues won't be attempted. These states correspond to the times when no process is executing the operation associated with them. Since there will never be an attempt to use them, no harm can be caused by defining a resulting state if they were used. Because of this fact, changes can be

made to the algorithm in chapter II which finds equivalent states and to the algorithm in chapter IV which converts states into multisets of elements.

When finding equivalent states, initially the states were divided into sets of similar states. When perilogues are used, this division should only be based on the prologues which may be applied. This is the same as allowing each epilogue to be applied at every state. When determining if two states within a set are related, any epilogue which is undefined at one of these states may be disregarded. Of course, this means that the relationship is no longer transitive. For example, if states p1, p2, and p3 are in set S1 and the epilogue for operation f is undefined at p1, takes p2 into set S2, and takes p3 into set S3, then p1 may be related to both p2 and p3 but p2 and p3 aren't related. When S1 is divided, p1 will be put in both of the new sets S4 and S5 containing p2 and p3 respectively. Now if some perilogue g takes a state p4 into p1, then p4 can be related to states which g takes into either S4 or S5. When this procedure is completed, a perilogue takes a set of states into each set into which it takes all of its member states. If it is an epilogue which is undefined for each state in the set, then it is undefined for the set. If there is a set of states T such that whenever a perilogue can result in T it also results in some other set, then T can be deleted. If a perilogue still takes a set into more than one resulting set, one of these resulting states is chosen.

When converting states into elements, it isn't important that the intersection of the states at which the epilogue of an operation may be applied not be contained in any other state. There is no problem if an epilogue can be applied at any of the other states. Thus, this check is only necessary for the prologues. A check still must be made to make sure that two states don't become equal.

REQUIRE AND RELEASE TRANSITIONS

While it was required that the epilogue must be able to be applied immediately upon completion of execution of an operation, it may be associated with more than one state transition. Thus, several may have to be tried before one is found that can be used. If the state transition used for the epilogue is unique given the one used for the prologue, then no search is necessary. In that case, all of the elements removed by this state transition must be included in every state in which the resource can be during the execution of the operation.

Definition: An epilogue is ***unique terminator*** if the state transition associated with it which is used at the end of execution of the corresponding operation is uniquely determined by the state transition which was used at the start of execution.

The epilogues for each operation in examples 5.2 and 5.3 may be seen to have only one final transition and therefore they trivially must be unique terminator.

For a state transition $U \rightarrow V$, as explained in chapter IV there are two reasons why an element might be in both U and V. The first is that it prevents several state transitions from being used in parallel. With concurrent resources, however, each state transition may be considered to be instantaneous and nothing else can happen while one is being used. The second reason is that this element is removed to prevent the state transition from being used at some state where it shouldn't be. If the state transition is associated with a unique terminator epilogue, though, it should be able to be used at any state at which it is attempted. Thus, if a state transition $U \rightarrow V$ is associated with a unique terminator epilogue, then U and V will be considered to be disjoint.

If an operation f has a unique terminator epilogue f_e and there is some element e such that only state transitions associated with f_e remove e, then e isn't needed. This can easily be seen since the presence or absence of e has no effect on whether a state transition associated with any other perilogue can be used at a state and it can only allow the state transitions associated with f_e to be used at more states than before. In example 5.2, c1, c2, and e can be deleted. In example 5.3, elements e5, e6, e7, e8, and e9 can be deleted.

It may now be observed that a state transition might not remove any elements or it might not add any elements.

Definition: A ***require transition*** is a state transition in which a set of elements is replaced by the empty set. A ***release transition*** is a state transition in which the empty set of elements is replaced by a set of elements.

A release transition may be used at every state. If one is associated with a prologue, then by theorem 4.4 there must be an infinite number of states. In examples 5.2 and 5.3, after each element is deleted which can be, every state transition associated with a prologue is a require transition and every one associated with an epilogue is a release transition.

A simplification to the implementation can be made when a require transition is used. If it is associated with a prologue and it was successfully used when the operation was called or else it is associated with an epilogue, then none of the waiting processes could execute before this transition so certainly none can execute after it and they don't need to be checked. If it is associated with the prologue of some operation called by a process which was blocked by the call, then any waiting lists

which have been checked during the current search and failed will still fail and therefore don't need to be checked again.

The following results help determine if it is possible for a state transition to be either a require or a release transition.

Theorem 5.4: If t:V → {} is a require transition which can be used at a state p, then any state transition t' which can be used at the resulting state p-V can also be used at p and t' commutes with t.

Proof: Let U = p-V. Let the state which results from using t' at U be q. Thus, using t' at p = U+V results in state q+V. Using t at this state must result in q.

Thus, if a require transition is associated with a prologue then every perilogue which can be used immediately after it is, except possibly the epilogue for that operation, must also be able to be used before it. Furthermore, they must commute. A similar result can also be shown for release transitions.

Theorem 5.5: If t:{} → V is a release transition which is used at a state p and t' can also be used at p with the result being state q, then t and t' commute.

Proof: The result of using t at p must be p+V, but since t' can be used at p, which is a substate of the new state, it can also be used there with the result being q+V. Using t at q also produces q+V.

Thus, if a release transition is associated with an epilogue, then every perilogue which can be used at some state where it can must commute with it.

MULTIPLE REGULAR EXPRESSIONS

If an operation is such that each state transition associated with its prologue is a require transition and each state transition associated with its epilogue is unique terminator and a release transition, then a set of elements is removed from the state of the resource at the start of execution and another set of elements is added at the completion. In this manner, each require transition associated with the prologue and the release transition of the epilogue which it uniquely determines may be united to form a state transition which may be associated with the operation itself like the state transitions which were used for serial resources.

Definition: An operation defined on a concurrent resource is ***united transitional*** if every state transition associated with its prologue is a require transition, the epilogue is unique terminator, and each state transition associated with the epilogue is a release transition. A concurrent resource is ***united transitional*** if every operation defined on it is.

United transitional resources have the advantage that the synchronization can be expressed in terms of the operations without concerning the programmer with prologues and epilogues. The concurrent resources of examples 5.2 and 5.3 are united transitional.

If a resource is united transitional, then the classifications described in chapter IV may be used. For instance, the resources of examples 5.2 and 5.3 are single transition since they are united transitional and the prologue of each operation is only associated with one state transition. In addition, every element in each of these resources has a multiplicity bounded by one. Thus, they are both boolean element. By theorem 4.11, the synchronization for the resources in these examples

may therefore be expressed using restricted multiple regular expressions. The expression for example 5.2 is $\{(f\ h)^*,(g\ h)^*\}$ and the expression for example 5.3 is $\{(p1+p2)^*,(p2+p3)^*,(p3+p4)^*,(p4+p5)^*,(p1+p5)^*\}$.

It may be seen by the above discussion that multiple regular expressions may be used to express the synchronization for some concurrent resources. It would be useful to know for exactly what class of synchronization they can be used. First, though, it is necessary to define what is meant by two perilogues being parallel at a state.

Definition: Two perilogues are *parallel* at a state p if they both may be applied there and they commute.

It is also necessary to introduce what is meant by two perilogues being sequential.

Definition: Perilogues f and g are *sequential* if any of the following are true.

1. There is a state where both f and g may be applied but at which they aren't parallel.
2. There are states p and q such that f takes p into q and g can be applied at q but not at p.
3. There is no state at which f and g are parallel.

It should be noted that a state was part of the definition of parallel perilogues but none was part of the definition of sequential perilogues. If there are states p and q at which perilogues f and g may both be applied, then it is possible that f and g commute at p but don't at q. Thus, they are parallel at p but are also sequential.

A natural restriction is to require that if two perilogues are parallel at some state then they aren't sequential. Parallelism may then be thought of as a symmetric relation.

Definition: A concurrent resource is *relationally parallel* if each of the following hold.

1. The prologue of each operation is parallel at each state p where it may be applied with every other perilogue which may be applied at the resulting state p' except for its epilogue.

2. The epilogue of each operation is parallel at each state p where it may be applied with every other perilogue which may be applied at p.

3. Sequential perilogues aren't parallel at any state.

4. The prologue and the epilogue of an operation can't both be applied at any state.

Conditions 1 and 2 basically insure that a relationally parallel resource must be united transitional. A proof that this is really true will be shown below. Condition 3 is explained above. Condition 4 requires that no two processes may simultaneously execute an operation. This may be seen from the following lemma.

Lemma 5.6: If a resource is relationally parallel then for each operation f the use of its prologue and its epilogue must alternate.

Proof: The prologue must be used before the epilogue. If the prologue f_p may be used at state p with the result being state q and $(q,g_1)...(q_{n-1},g_n)$ is an arc progression such that $(\forall i,\ 1 \le i < n)\ g_i \neq f_e$, then $(\forall i,\ 1 \le i < n)$ f_e can be applied at q_i and therefore f_p can't be. Also, f_p can't be applied at q. Thus, $(\forall i,\ 1 \le i \le n)\ g_i \neq f_p$.

It is now possible to show that there are some finite state concurrent resources for which the synchronization can't be expressed using a multiple regular expression.

Theorem 5.7: Every concurrent resource for which the synchronization can be expressed using a multiple regular expression is both united transitional and relationally parallel.

Proof: A multiple regular expression may be implemented by converting each of the member expressions into a finite automaton, each with a disjoint set of states. The resource state will be represented by one state from each of these automata. When an operation starts executing, each of the automata corresponding to expressions in which it occurs must be in a state at which the operation may be applied. The states of these automata are set to the null state until the operation finishes and then are each set to new states based on the starting states. Thus, it may be seen that every state transition associated with the prologue is a require transition, each associated with the epilogue is a release transition, and the epilogue is unique terminator. Thus, the resource is united transitional and by theorems 5.4 and 5.5 conditions 1 and 2 of the definition of relationally parallel hold. If perilogues for two operations are parallel at some state, then both operations may be able to execute concurrently and they can't both be included in the same expression. If the prologues for both operations may both be applied at the same state and they aren't parallel, then by theorem 5.4 neither prologue may immediately follow the other. Thus, they must compete for the state of one of the automata which can't be true since they aren't in the same expressions. Condition 4 holds since the epilogue of an operation can only be applied when each of the appropriate automata are in the null state and the prologue can't be applied then.

PROCEDURES

It is sometimes necessary that several operations defined on a resource perform the same suboperation. For example, if there are operations defined on a stack to pop the top element off and another to return the top element but to leave it on the stack, both operations must first test to see that the stack isn't empty. It is standard programming practice to use a procedure for this purpose.

For a serial resource, no two operations may execute simultaneously so there can be at most one call on the procedure in progress at any given time. Any other synchronization of the procedure must also be contained in the synchronization of the calling operations. This is because the procedure itself can't be included as an operation in the synchronization of the resource. Otherwise, since the calling operation is executing, the call will cause the procedure to block. The result is a deadlock.

An alternative method which can be used for serial resources which allows procedures to be synchronized is to define the operations as sequences of procedures. For example, if an operation f uses a procedure g, it might be written as f = sf;g;ff where sf and ff are also procedures and can be included in the synchronization of the resource along with g.

For a concurrent resource it may be possible for two operations to simultaneously call a procedure. If only one invocation of the procedure can be allowed at a time, it must be included in the synchronization for the resource. In this case, it must be possible for both the calling operation and the procedure to be

executed at the same time. This will only be true if there is a composed state transition from each state which can result from the prologue of a calling operation to a state at which the procedure can be applied. Also, these composed state transitions shouldn't contain the final transition of the calling operation. If this condition doesn't hold, it will be possible for a calling operation to start and become deadlocked when the procedure is called.

When an operation calls a procedure which is synchronized, it is possible to drop the restriction discussed earlier that the final transition of an operation can be applied at every state which can occur during the execution of the operation. In this case, the restriction only needs to be enforced for all of the states which the resource can be in after the last such procedure has been executed since the operation can't complete until this occurs.

CHAPTER VI

CONCLUSION

In this research, the problem of synchronizing operations defined on a shared resource was studied. The approach was to express the sequences of operations which are allowed on the resource by creating synchronization relationships consisting of a group of states and a successor function. An alternative model was also given in which states were represented as multisets of elements and the state changes caused by the execution of an operation were expressed as state transitions.

A series of restrictions to this model was presented to isolate classes of synchronization due to implementation or notation. The first restriction was that only those resources for which the synchronization could be expressed using a finite number of states was studied. The next restriction was to require that the successor function be injective with respect to each operation. Another class, called single transition, was shown to be a subclass of the injective resources. A further restriction of the single transition resources produced the boolean element resources. A subclass of the boolean element resources was formed by requiring that the successor function be projective with respect to each operation. These resources were called simple serial. The final restriction was to disallow Z expressions from the initial state to any persistent set entry state. This leads to the restricted regular expressions described in chapter III.

An open question which was left unanswered was the characterization of the

single transition resources in terms of restrictions on the successor function. These resources were shown to be injective and several other properties were shown to hold. However, these restrictions aren't sufficient. The problem is that the restriction is dependent on the sequences in which an operation can occur rather than just on the states at which it may be used.

An extension was made to this model to allow concurrent resources to be handled. It was shown that synchronization couldn't be expressed as sequences of the operations but that prologues and epilogues were needed. Three different levels of systems were looked at. The first restricts the model to only a finite number of states. The next allows only relationally parallel resources. The final restriction also requires that the resource be boolean element.

The method which can be used to implement synchronization was shown to be increasingly simple as the model was restricted. Implementations were given for all finite state resources, the single transition resources, boolean element resources, and simple serial resources. The overhead required to decide if an operation could start execution was discussed. The differences between one waiting list and several in relation to the difficulty of a search when an operation completes execution was also shown.

The problem of deadlocks was briefly discussed when it was shown that every permanent operation must be an auxiliary of every persistent set. However, this won't prevent deadlocks from occurring. They are also dependent on the sequences of calls made by each process. Consider, for example, two resources with synchronization specified by the restricted regular expressions $(e\ f)^*$ and $(g\ h)^*$ respectively. Assume

that these resources are used by two processes, one of which calls f and then g and the other which calls h and then e. A deadlock will result. In general, the solution to this problem isn't computable. Even if the processes are restricted such that it is decidable whether or not a deadlock will occur, it would involve checking the code of each process which uses some shared resource.

One of the reasons for this study was to provide a means with which various methods of synchronization could be classified. It was shown that the class of resources which can be synchronized by using boolean semaphores such that at most one may be positive at a time and requiring that an operation do a P on one before it starts and a V on one when it completes corresponds to the simple serial resources. Allowing P-V multiple but still allowing only boolean semaphores corresponds to the boolean element resources. Finally, combining P-V multiple and P-V chunk such that an operation may start by decrementing several semaphores by values which may be greater than one but requiring that the semaphores be bounded corresponds to the single transition operations.

Various forms of regular expressions were also looked at. Restricted regular expressions were shown to correspond to the simple serial resources without Z expressions, restricted multiple regular expressions were shown to correspond to the boolean element resources, regular expressions themselves were shown to correspond to the finite state resources, and multiple regular expressions were shown to be a subset of the relationally parallel resources.

A possibility for further work is to study other modifications to regular expressions. This would involve discovering the necessary restrictions to the model

and then proving that the new form of regular expression and the restriction represent the same synchronization. One possibility is the parallel regular expression which allows the notation R//R' where R and R' are regular expressions. The meaning is that a sequence of operations allowed by R and a sequence of operations allowed by R' can be executed concurrently. The advantage over multiple regular expressions is that all of the synchronization is contained in one expression and not spread across several, thus improving understandability. There is also a disadvantage in that some synchronization which can be expressed using restricted multiple regular expressions can't be expressed using parallel regular expressions. For example, consider $\{(a\ c)^*,(a\ d)^*,(b\ c)^*\}$. Initially, a and b can be executed simultaneously. After they have each finished, c can be executed and then a and b again. This can be expressed with $((a//b)c)^*$. It isn't possible to add d to this expression so that it follows a and executes in parallel with b and c. It might also be desirable to restrict these expressions in some manner similar to restricted regular expressions in order to simplify the implementation.

Another way in which regular expressions could be altered would be to allow parameters in some manner. For example, the size of a stack influences the number of states needed to synchronize the operations PUSH and POP and therefore it affects the regular expression used. When a type STACK is defined, it shouldn't be restricted to a specific size. This decision should be postponed until a specific instance is declared. One suggestion [Ha75] is to allow the notation $(f\text{-}g)^n$. This is similar to the notation f+g except that the number of times that f has been executed at any given time minus the number of times that g has executed must be neither negative nor greater than n. Thus, the synchronization for the stack can be expressed using the expression $((PUSH\text{-}POP)^n)^*$.

Another extension of the work described here is to allow an infinite number of states. Considering each state and listing the values for the successor function for such a resource is impossible. However, using elements to construct states and associating operations with state transitions yields a possible solution to this problem. It would still be necessary to restrict the resources such that there are only a finite number of elements and each operation can only be associated with a finite number of state transitions. This is a reasonable restriction since most infinite state resources, such as an unbounded stack, are usually implemented using an unbounded counter or semaphore which is then replaced by an element in the model.

The use of modified regular expressions in a resource definition to describe synchronization is an attempt to make this synchronization more understandable and the resource more reliable. Hopefully, a high level programming language containing some form of regular expressions, which was selected based on this study, will be developed. This research could then be considered to have made a small contribution to the area of reliable software.

BIBLIOGRAPHY

[AU72] Aho, A. V. and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol 1: Parsing, Prentice-Hall, Inc, Englewood Cliffs, NJ, 1972.

[B74] Bekkers, Y., "A Comparison of Two High Level Synchronizing Concepts," Queen's University of Belfast Report, 1974.

[BH72] Brinch Hansen, P., "Structured Multiprogramming," CACM 15,7 (July 1972), pp. 574-578.

[CHa74] Campbell, R. H. and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," International Symposium on Operating Systems Theory and Practice, IRIA, Paris, April 1974, pp. 93-106.

[CHP71] Courtois, P. J., F. Heymans, D. L. Parnas, "Concurrent Control with "Readers" and Writers"," CACM 14,10 (Oct 1971), pp. 667-668.

[D68] Dijkstra, E. W., "Cooperating Sequential Processes," Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, pp. 43-112.

[F75] Flon, L., "Program Design with Abstract Data Types," Carnegie-Mellon University Report, June, 1975.

[Ha72] Habermann, A. N., "Synchronization of Communicating Processes," CACM 15,3 (March 1972), pp. 171-176.

[Ha75] Habermann, A. N., "On the Timing Restrictions of Concurrent Processes," Fourth Texas Conference on Computing Systems, Austin, TX, 1975.

[Ho74] Hoare, C. A. R., "Monitors: An Operating System Concept," CACM 17,10 (Oct 1974), pp. 549-557.

[HU69] Hopcroft, J. E. and J. D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley Publishing Co, Reading, MA, 1969.

[K69] Knuth, D. E., The Art of Computer Programming, Vol 2: Seminumerical Algorithms, Addison-Wesley Publishing Co, Reading, MA, 1969.

[L73] Lipton, R. J., "On Synchronization Primitive Systems," Ph.D. Thesis, Carnegie-Mellon University, June 1973.

[P71] Patil, S. S., "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes," Project MAC, Computational Structures Group Memo 57, Feb 1971.

[PKT73] Peterson, W. W., T. Kasami, and N. Tokura, "On Capabilities of While, Repeat, and Exit Statements," CACM 16,8 (Aug 1973) pp. 503-512.

[VL72] Vantilborgh, H. and A. van Lamsweerde, "On an Extension of Dijkstra's Semaphore Primitives," Information Processing Letters 1 (1972), pp. 181-186.

[W72] Wodon, P., "Still Another Tool for Synchronizing Cooperating Processes," Carnegie-Mellon University Report, Aug 1972.

APPENDIX

This appendix contains proofs of theorems from chapter III.

Theorem 3.1: For a restricted regular expression R with the corresponding finite automaton (K,Σ,δ,p,F) the following properties are true.

Property 3.1.1: Either R is final loop or $(\forall q \in F)\ (\forall f \in \Sigma)\ \delta(q,f)$ is dead.

Property 3.1.2: $(\forall f \in \Sigma)\ (\forall q,q' \in K)$ either $\delta(q,f)$ or $\delta(q',f)$ is dead.

Property 3.1.3: $(\exists f \in \Sigma)\ \delta(p,f)$ isn't a dead state.

Property 3.1.4: Either R is initial loop or $(\forall q \in K)\ (\forall f \in \Sigma)\ \delta(q,f) \neq p$ and p isn't in F.

Property 3.1.5: If R is simple (not selection) final loop, then there is only one state in F.

Furthermore, let (K,Σ,δ,p,F) be the finite automaton corresponding to R and $(K',\Sigma',\delta',p',F')$ correspond to R'. Then $(K\text{-}F,\Sigma,\delta_1,p,\{p\})$ corresponds to R^*, either $((K\text{-}F)\cup K',\Sigma\cup\Sigma',\delta_2,p,F')$ corresponds to RR' or there is only one state $p'' \in F$ and $(K\cup(K'\text{-}\{p'\}),\Sigma\cup\Sigma',\delta_3,p,F')$ corresponds to RR', and $(K\cup(K'\text{-}\{p'\}),\Sigma\cup\Sigma',\delta_4,p,F\cup F')$ corresponds to $R+R'$ where

$\delta_1(q,f)$	$= p$	if $\delta(q,f) \in F$
	$= \delta(q,f)$	otherwise $(\forall q \in (K\text{-}F))$
$\delta_2(q,f)$	$= p'$	if $\delta(q,f) \in F$
	$= \delta(q,f)$	otherwise $(\forall q \in (K\text{-}F))\ (\forall f \in \Sigma)$
	$= \delta'(q,f)$	$(\forall q \in K')\ (\forall f \in \Sigma')$
$\delta_3(p'',f)$	$= \delta'(p',f)$	$(\forall f \in \Sigma')$
$\delta_3(q,f)$	$= \delta(q,f)$	$(\forall q \in K)\ (\forall f \in \Sigma)$
	$= \delta'(q,f)$	$(\forall q \in (K'\text{-}\{p'\}))\ (\forall f \in \Sigma')$
$\delta_4(p,f)$	$= \delta'(p',f)$	$(\forall f \in \Sigma')$
$\delta_4(q,f)$	$= \delta(q,f)$	$(\forall q \in K)\ (\forall f \in \Sigma)$
	$= \delta'(q,f)$	$(\forall q \in (K'\text{-}\{p'\}))\ (\forall f \in \Sigma')$

Any arguments for which δ_2, δ_3, or δ_4 are undefined are dead.

Lemma 3.1.6: If R^* is a restricted regular expression such that R and the corresponding finite automaton (K,Σ,δ,p,F) satisfy properties 3.1.1 to 3.1.4 then the finite automaton corresponding to R^* is $(K\text{-}F,\Sigma,\delta',p,\{p\})$ where

$$\begin{aligned} \delta'(q,f) &= p && \text{if } \delta(q,f) \in F \\ &= \delta(q,f) && \text{otherwise } (\forall q \in (K\text{-}F)) \end{aligned}$$

and properties 3.1.1 to 3.1.5 are satisfied.

Proof: By the definition of a restricted regular expression, R is neither initial nor final loop. Thus, by property 3.1.1 $(\forall q \in F)$ $(\forall f \in \Sigma)$ $\delta(q,f)$ is dead and by property 3.1.4 p isn't in F. As shown in chapter III, $(\forall q \in F)$ $(\forall f \in \Sigma)$ $\delta'(q,f) = \{\delta(p,f),\delta(q,f)\}$. But $\delta(q,f)$ is a dead state and entering such a state will never result in acceptance of the string. Therefore, $\delta'(q,f) = \delta(p,f)$ and q and p' are equivalent and since both p' and q are final states, they can be combined, leaving p' as the only final state. Since R isn't initial loop, by property 3.1.4 $(\forall q \in K)$ $(\forall f \in \Sigma)$ $\delta(q,f) \neq p$. Thus, p can be deleted and p' renamed to be p. Properties 3.1.1 and 3.1.4 are true for R^* since it is both initial and final loop. Property 3.1.2 is true since $(\forall f \in \Sigma)$ if $(\exists q \in K\text{-}F)$ $\delta(q,f) \in F$ then $(\forall q' \neq q)$ $\delta'(q',f) = \delta(q',f)$ is dead. Otherwise $(\forall q,q' \in K\text{-}F)$ either $\delta'(q,f) = \delta(q,f)$ or $\delta'(q',f) = \delta(q',f)$ is dead. Since property 3.1.3 is true for R $(\exists f \in \Sigma)$ $\delta(p,f)$ isn't a dead state. Either $\delta'(p,f) = p$ or $\delta'(p,f) = \delta(p,f)$, so $\delta'(p,f)$ isn't dead and property 3.1.3 is true for R^*. Property 3.1.5 is trivially true since p is the only final state.

Lemma 3.1.7: Let RR' be a restricted regular expression such that R and the corresponding finite automaton (K,Σ,δ,p,F) and R' and its corresponding finite automaton $(K',\Sigma',\delta',p',F')$ satisfy properties 3.1.1 to 3.1.5. Let M be the finite automaton corresponding to RR'. If R is final loop, then $M = (K \cup (K'-\{p'\}),\Sigma \cup \Sigma',\delta'',p,F')$ and $F = \{p''\}$ where

$$\begin{aligned}
\delta''(p'',f) &= \delta'(p',f) && (\forall f \in \Sigma')\\
\delta''(q,f) &= \delta(q,f) && (\forall q \in K)\ (\forall f \in \Sigma)\\
&= \delta'(q,f) && (\forall q \in (K'-\{p'\}))\ (\forall f \in \Sigma')
\end{aligned}$$

and $\delta''(q,f)$ is a dead state for all other (q,f) pairs. Otherwise, $M = ((K-F)\cup K', \Sigma\cup\Sigma', \delta'', p, F')$ where

$$\begin{aligned}
\delta''(q,f) &= p' && \text{if } \delta(q,f) \in F\\
&= \delta(q,f) && \text{otherwise } (\forall q \in (K-F))\ (\forall f \in \Sigma)\\
&= \delta'(q,f) && (\forall q \in K')\ (\forall f \in \Sigma')
\end{aligned}$$

and $\delta''(q,f)$ is a dead state for all other (q,f) pairs. In either case, RR' and M satisfy properties 3.1.1 to 3.1.5.

Proof: By the definition of a restricted regular expression, Σ and Σ' are disjoint. Therefore, as shown in chapter III, $(\forall q \in K)\ (\forall f \in \Sigma)\ \delta''(q,f) = \delta(q,f)$, $(\forall q \in F)\ (\forall f \in \Sigma')\ \delta''(q,f) = \delta'(p',f)$, $(\forall q \in K')\ (\forall f \in \Sigma')\ \delta''(q,f) = \delta'(q,f)$, and $\delta''(q,f)$ is a dead state for all other (q,f) pairs. If R isn't final loop, then by property 3.1.1 $(\forall q \in F)\ (\forall f \in \Sigma)\ \delta(q,f)$ is dead. Thus, $(\forall q \in F)\ (\forall f \in \Sigma\cup\Sigma')\ \delta''(q,f) = \delta''(p',f)$ and q is a final state iff p' is. Therefore, F can be deleted and $(\forall q' \in (K-F))\ (\forall f \in \Sigma)$ if $\delta(q',f) \in F$ then $\delta''(q',f) = p'$. If R is final loop, then R' can't be initial loop. By property 3.1.4 $(\forall q \in K')\ (\forall f \in \Sigma')\ \delta'(q,f) \neq p'$ and p' isn't in F'. Therefore, $F'' = F'$ and $(\forall q \in K\cup K')\ (\forall f \in \Sigma\cup\Sigma')\ \delta''(q,f) \neq p'$ so p' can be deleted. Property 3.1.1 holds since if R' is final loop, then so is RR'. Otherwise, by property 3.1.1 $(\forall q \in F')\ (\forall f \in \Sigma')\ \delta'(q,f) = \delta''(q,f)$ is dead. Also, $(\forall q \in F')\ (\forall f \in \Sigma)\ \delta''(q,f)$ is dead. Property 3.1.2 must hold since if R is final loop it must be simple final loop and by property 3.1.5 there is only one state p'' in F. $(\forall f \in \Sigma)\ (\forall q \in (K'-\{p'\}))\ \delta''(q,f)$ is dead and $(\forall q,q' \in K)$ either $\delta''(q,f) = \delta(q,f)$ or

$\delta''(q',f) = \delta(q',f)$ is dead. $(\forall g \in \Sigma')\ (\forall q \in K-\{p''\})\ \delta''(g,q)$ is dead. If $\delta''(p'',g) = \delta'(p',g)$ isn't dead, then $(\forall q \in K'-\{p'\})\ \delta''(q,g) = \delta'(q,g)$ is dead. Otherwise, $(\forall q,q' \in K'-\{p'\})$ either $\delta''(q,g) = \delta'(q,g)$ is dead or else $\delta''(q',g) = \delta'(q',g)$ is dead. If R isn't final loop, then $(\forall f \in \Sigma)\ (\forall q \in K')\ \delta''(q,f)$ is dead and $(\forall q,q' \in K-F)$ either $\delta(q,f)$ is dead in which case eitner $\delta''(q',f)$ is dead and $\delta''(q',f) = \delta(q',f)$ or else $\delta''(q,f) = \delta(q,f)$. Likewise, $(\forall f \in \Sigma')\ (\forall q \in K-F)\ \delta''(q,f)$ is dead and $(\forall q,q' \in K')$ either $\delta''(q,f) = \delta'(q,f)$ is dead or else $\delta''(q',f) = \delta'(q',f)$ is dead. Property 3.1.3 holds since either $\delta''(p,f) = p'$ or $\delta''(p,f) = \delta(p,f)$. By property 3.1.3, $(\exists f \in \Sigma)\ \delta(p,f)$ isn't dead. If R is initial loop then so is RR'. Otherwise, by property 3.1.4 p isn't in F and $(\forall q \in K)\ (\forall f \in \Sigma)$ $\delta(q,f) \neq p$. Since $(\forall q \in K')\ (\forall f \in \Sigma')\ \delta'(q,f) \neq p$, $(\forall f \in \Sigma \cup \Sigma')\ (\forall q \in K \cup K')$ $\delta''(q,f) \neq p$. Thus, property 3.1.4 holds for RR'. If RR' is simple final loop then so is R' and there can only be one state in F'.

Lemma 3.1.8: Let R+R' be a restricted rugular expression such that R and the corresponding finite automaton (K,Σ,δ,p,F) and R' and its corresponding finite automaton $(K',\Sigma',\delta',p',F')$ satisfies properties 3.1.1 to 3.1.4. The finite automaton corresponding to R+R', $(K \cup (K'-\{p'\}),\Sigma \cup \Sigma',\delta'',p,F \cup F')$ where

$$\begin{array}{lll} \delta''(p,f) & = \delta'(p',f) & (\forall f \in \Sigma') \\ \delta''(q,f) & = \delta(q,f) & (\forall q \in K)\ (\forall f \in \Sigma) \\ & = \delta'(q,f) & (\forall q \in (K'-\{p'\}))\ (\forall f \in \Sigma') \end{array}$$

and $\delta''(q,f)$ is a dead state for all other (q,f) pairs, satisfy properties 3.1.1 to 3.1.5.

Proof: Neither R nor R' are initial loop, so by property 3.1.4 $(\forall q \in K)$ $(\forall f \in \Sigma)\ \delta(q,f) \neq p$ and $(\forall q' \in K')\ (\forall g \in \Sigma')\ \delta'(q',g) \neq p'$. Therefore, both p and p' can be deleted. Also, p isn't in F and p' isn't in F' and therefore p"

isn't in F". By the definition of a restricted regular expression, Σ and Σ' are disjoint. Thus, as shown in chapter III, $(\forall f \in \Sigma)\ \delta''(p'',f) = \delta(p,f)$ and $(\forall g \in \Sigma')\ \delta''(p'',g) = \delta'(p',g)$. Since p was deleted, p" can be renamed to be p. Property 3.1.4 holds since by the definition of restricted regular expression, neither R nor R' is initial loop and by property 3.1.4, $(\forall q \in K \cup (K'-\{p'\}))\ (\forall f \in \Sigma \cup \Sigma')\ \&*''(q,f) \neq p$. Property 3.1.1 holds since if either R or R' is final loop, then so is R+R'. Otherwise, by property 3.1.1 $(\forall q \in F)\ (\forall f \in \Sigma)\ \delta''(q,f) = \delta(q,f)$ is dead. Likewise, $(\forall q \in F')\ (\forall f \in \Sigma')$ $\delta''(q,f) = \delta'(q,f)$ is dead. Property 3.1.2 holds since $(\forall f \in \Sigma)\ (\forall q \in K')\ \delta''(q,f)$ is dead and $(\forall q' \in K)\ \delta''(q',f) = \delta(q',f)$, $(\forall q,q' \in K \cup (K'-\{p'\}))$ either $\delta''(q,f)$ or $\delta''(q',f)$ is dead. Also, $(\forall g \in \Sigma')\ (\forall q \in K-\{p\})\ \delta''(q,g)$ is dead. If $\delta''(p,g) = \delta'(p',g)$ isn't dead, then $(\forall q' \in K'-\{p'\})\ \delta''(q',g) = \delta'(q',g)$ is dead. Otherwise, $\delta''(p,g)$ is dead and $(\forall q,q' \in K'-\{p'\})$ either $\delta''(q,g) = \delta'(q,g)$ or $\delta''(q',g) = \delta'(q',g)$ is dead. By property 3.1.3 $(\exists f \in K)\ \delta''(p,f) = \delta(p,f)$ isn't dead. Property 3.1.5 holds since R+R' can't be simple final loop.

The proof of theorem 3.1 will now be given.

Proof: The proof is by induction on the complexity of the regular expression. For the finite automaton $(\{p,q,q'\},\{f\},\delta,p,\{q\})$ where $\delta(p,f) = q$ and $\delta(q,f) = \delta(q',f) = q'$, $\delta(q,f)$ and $\delta(q',f)$ are dead and $\delta(p,f)$ isn't, so properties 3.1.1, 3.1.2, and 3.1.3 are true. Also, there is only one final state which isn't p and there is no state p' such that $\delta(p',f) = p$. Thus, from lemmas 3.1.6, 3.1.7, and 3.1.8, a finite automaton can be constructed as indicated in the theorem and the properties hold.

Theorem 3.5: A shared resource on which the allowable sequences of operations are given by a restricted regular expression is simple serial with no Z expression from the initial state to a final state.

Lemma 3.5.1: Assume that the synchronization for a resource is expressed by the restricted regular expression R^* and that the synchronization for R has no Z expression from the initial state p to a final state. Then the synchronization for R^* doesn't have a Z expression from p to p either.

Proof: If there is a Z expression from p to p then by lemma 3.2 there also is a simple Z expression $a_1...a_k$ from p to q, $b_1...b_m$ from q' to q, and $c_1...c_n$ from q' to p such that $q \neq p$. Also, by the definition of a simple Z expression, $(\forall q")$ if $(\exists f \in \Sigma)$ such that either $(\exists i,\ 1<i\leq k)\ (q",f) = a_i$, $(\exists i,\ 1<i\leq m)\ (q",f) = b_i$, or $(\exists i,\ 1<i\leq n)\ (q",f) = c_i$ then $q" \neq p$. Thus, $(\forall i,\ 1\leq i\leq k)$ $\delta_1(a_i) \neq p$ and therefore $\delta_1(a_i) = \delta(a_i)$, $(\forall i,\ 1\leq i\leq m)$ $\delta_1(b_i) \neq p$ and therefore $\delta_1(b_i) = \delta(b_i)$, and $(\forall i,\ 1\leq i<n)$ $\delta_1(c_i) \neq p$ and therefore $\delta_1(c_i) = \delta(c_i)$. By property 4 of theorem 3.1, since R can't be initial loop by the definition of a restricted regular expression, $(\forall q \in K)\ (\forall f \in \Sigma)\ \delta(q,f) \neq p$. Thus, $\delta(c_n) \neq p$ and it must be true that $\delta(c_n) \in F$. Therefore, $a_1...a_k$, $b_1...b_m$, and $c_1...c_n$ form a Z expression from p to some element of F in R, a contradiction.

Lemma 3.5.2: Assume that the synchronization for a resource is expressed by the restricted regular expression R+R' and that neither the synchronization for R nor for R' has a Z expression from the initial state p or p' respectively to a final state. Then the synchronization for R+R' doesn't have a Z expression from p to a final state either.

Proof: Assume that there is a Z expression $\alpha = (p,g_1)(p_1,g_2)...(p_{n-1},g_n)$, β from q to p_n, and γ from q to $q' \in (F \cup F')$ in R+R'. By the definition of a restricted regular expression, R+R' can't be initial loop and by property 3.1.4 $(\forall q" \in K \cup (K'-\{p'\}))\ (\forall f \in \Sigma \cup \Sigma')\ \delta_4(q",f) \neq p$. Let $K_1 = K-\{p\}$ and

$K_2 = K'-\{p'\}$. Thus, $p_1 \in K_1$ or $p_1 \in K_2$. Assume that $p_1 \in K_1$. By the definition of δ_4, $(\forall i,\ 1 \le i \le n)\ p_i \in K_1$. If $q \in K_2$, then $(\forall (q'',f)$ in $\beta)\ q'' \in K_2$ and $p_n \in K_2$, a contradiction. Thus, $q \in K_1$ and $(\forall (q'',f)$ in $\beta)\ q'' \in K_1$ and $f \in \Sigma$. Likewise, $(\forall (q'',f)$ in $\gamma)\ q'' \in K_1$, $f \in \Sigma$, and $q' \in K_1$. Since q' is also in $F \cup F'$, α, β, and γ forms a Z expression from p to $q' \in F$ in R. If $p_1 \in K_2$, then $g_1 \in \Sigma'$ and $\delta'(p',g_1) = p_1$. Using an argument similar to the one above, it may be shown that if $p_1 \in K_2$ then α, β, and γ form a Z expression from p' to $q' \in F'$ in R'.

Lemma 3.5.3: Assume that the synchronization for a resource is expressed by the restricted regular expression RR' and that neither the synchronization for R nor for R' has a Z expression from the initial state p or p' respectively to a final state. Then the synchronization for RR' doesn't have a Z expression from p to a final state either.

Proof: There is no arc progression $(q_0,f_1)\ldots(q_{n-1},f_n)$ from a state $q_0 \in K'$ to $q_n \in K'$. Otherwise, $(\exists i,\ 0 \le i < n)\ q_i \in K'$ and $(\forall j,\ i < j \le n)\ q_j \in K$. Thus, $\delta_2(q_i,f_{i+1}) = q_{i+1} \in K$ or $\delta_3(q_i,f_{i+1}) = q_{i+1}$ which contradicts the definition of δ_2 and δ_3. If there is an arc progression $\alpha = (q_0,f_1)\ldots(q_{n-1},f_n)$ from $q_0 \in K$ to $q_n \in K'$ then $(\exists i,\ 0 < i \le n)\ q_i \in K'$ and $(\forall j,\ 0 \le j < i)\ q_j \in K$. It must also be true that $(\forall j,\ i < j \le n)\ q_j \in K'$. Otherwise, there is an arc progression from $q_i \in K'$ to $q_j \in K$. Assume there is a Z expression from p to a state $q'' \in F'$. Then there is a simple Z expression $a_1\ldots a_k$ from p to q, $b_1\ldots b_m$ from q' to q, and $c_1\ldots c_n$ from q' to q". It can't be true that $q' \in K'$ and $q' \in K$. Thus, either $q \in K'$ or $q' \in K$.

Case 1: R isn't final loop and δ_2 is used. If there is such an α then $q_i = p'$. If $q \in K'$ then $(\exists j1,\ 1 < j1 \le k)\ (\exists f \in \Sigma')\ a_{j1} = (p',f)$. If $q' \in K$ then $(\exists j2,\ 1 < j2 \le n)\ (\exists f \in \Sigma')\ c_{j2} = (p',f)$. It must also be true that $\delta(c_{j2-1}) = p_F \in F$. If $q \in K'$ and $q' \in K$ then $(\exists j,\ 1 < j \le m)\ (\exists f \in \Sigma')\ b_j = (p',f)$, a contradiction of the definition of a Z expression. If $q' \in K'$

then ($\forall i$, $1 \le i \le m$) if $b_i = (p'',f)$ then $p'' \in K'$ and $f \in \Sigma'$. Also, ($\forall i$, $1 \le i \le n$) if $c_i = (p'',f)$ then $p'' \in K'$ and $f \in \Sigma'$. Thus, $a_{j1} \ldots a_k$, $b_1 \ldots b_m$, and $c_1 \ldots c_n$ form a Z expression from p' to q'' in R'. Note that by the definition of a simple Z expression there is no $b_i = (p',f)$. Similarly, if $q \in K$ and $q' \in K$ then $a_1 \ldots a_k$, $b_1 \ldots b_m$, and $c_1 \ldots c_{j2-1}$ form a Z expression from p to p_F in R.

Case 2: R is simple final loop, δ_3 is used, and there is only one state $p'' \in F$. If there is such an α then $q_{i-1} = p''$. If $q \in K'$ then ($\exists j3$, $1 \le j3 < k$) ($\exists f3 \in \Sigma'$) $a_{j3} = (p'',f3)$. If $q' \in K$ then ($\exists j4$, $1 \le j4 < n$) ($\exists f4 \in \Sigma'$) $c_{j4} = (p'',f4)$. It must also be true that $\delta'(p',f4) = \delta_3(p'',f4)$. If $q \in K'$ and $q' \in K$ then ($\exists j$, $1 \le j < m$) ($\exists f \in \Sigma'$) $b_j = (p'',f)$, a contradiction of the definition of a Z expression. If $q' \in K$, then $a_1 \ldots a_k$, $b_1 \ldots b_m$, and $c_1 \ldots c_{j4-1}$ must be a Z expression from p to p'' in R. If $q \in K'$, then $(p',f)a_{j3+1} \ldots a_k$, $b_1 \ldots b_m$, and $c_1 \ldots c_n$ form a Z expression from p' to q'' in R'.

The proof of theorem 3.5 will now be given.

Proof: By property 3.1.2, the resource must be simple serial. There clearly is no Z expression from the initial state to a final state of a single operation expression. Thus, from lemmas 3.5.1, 3.5.2, and 3.5.3 the theorem must hold.

Theorem 3.7: A simple serial resource with no Z expression from the initial state to a state q such that either no operation may be applied at it or else q is a persistent set entry state can be written as a restricted regular expression without repeated names.

Definition: The *final states* of a simple serial resource as described in the theorem are the persistent set entry states and the states at which no operation can be applied.

Lemma 3.7.1: Assume the synchronization for a finite state resource M with initial state p has the property that there is an arc progression from every state to p. Let M' be a resource which differs from M in that there is an additional state p' at which no operations can be applied and the successor function S' is defined as follows: S'(q,f) = p' if S(q,f) = p and S'(q,f) = S(q,f) otherwise. If M is simple serial with no Z expression from p to p then M' is simple serial with no Z expression from p to p'. Also, if the synchronization for M' can be expressed with the regular expression R then the synchronization for M can be expressed with R^*. Finally, there is no persistent set in M'.

Proof: An operation can be applied at a state of M iff it can be applied at the same state of M'. Thus, M is simple serial iff M' is. Since there is an arc progression from every state of M to p, at least one operation can be applied at every state of M and therefore also at every state of M' except for p'. Every arc progression in M to p is an arc progression in M' to p'. Hence there can be no persistent set and p' is the only final state. By theorem 3.1, if the synchronization for M' can be represented by R, then the synchronization for M can be represented by R^*. Finally, assume that there is a Z expression from p to p' in M'. By the definition of M', this must also be a Z expression from p to p in M.

Lemma 3.7.2: For a finite state resource with initial state p, if there is an arc progression from some state q to p but none from another state q' to p then there can be no arc progression from q' to q. Furthermore, if there is no Z expression from p to a final state, then every arc progression from q to q' must contain an arc (p,f).

Proof: If there is an arc progression from q' to q, since there is one from q to p, there must be an arc progression from q' to p, a contradiction. If $(q_0,f_1)...(q_{n-1},f_n)$ is an arc progression from $q = q_0$ to $q' = q_n$, then ($\exists k$, $0 \leq k < n$) there is an arc progression α from q_k to p but there is no arc progression from q_{k+1} to p. There must also be an arc progression β from q_{k+1} to some final state q_f. Since there is no arc progression from q_{k+1} to p, there can be no arc (p,f) in β and there can be no arc (q_f,f) in α. If $q_k \neq p$, then ϵ, α, and β form a Z expression from p to q_f, violating the assumption. Therefore, $q_k = p$ and there is an arc (p,f_{k+1}) in the arc progression from q to q'.

Lemma 3.7.3: If there is a Z expression from a state p to itself then ($\forall q$) if there is an arc progression from p to q then there is a Z expression from p to q.

Proof: Since there is a Z expression from p to p, by lemma 3.2 there is also a simple Z expression from p to p composed of arc progressions $\alpha = (p_0,g_1)...(p_{m-1},g_m)$ from $p = p_0$ to some state q_n, $\beta = (q_0,f_1)...(q_{n-1},f_n)$ from a state q_0 to q_n, and γ from q_0 to p such that ($\forall i$, $0<i<n$) $q_i \neq p$, ($\forall j$, $0<j\leq n$) there is no arc (q_j,f) in γ, ($\forall k$, $0\leq k<n$) there is no arc (q_k,f) in α, there is no arc (p,f) in γ, and $p \neq q_0$. There are several cases which must be handled. In each, ϵ represents the empty arc progression.

Case 1: If $p = q$, then there trivially is a Z expression from p to q.

Case 2: If ($\exists k$, $0<k\leq n$) $q_k = q$, then there are no arcs (p,f) or (q_k,f) in γ, $p \neq q = q_k$, $p \neq q_0$, and ($\forall i$, $0<i<k\leq n$) $p \neq q_i$. Thus, ϵ, γ, and $(q_0,f_1)...(q_{k-1},f_k)$ form a Z expression from p to q.

Case 3: If there is an arc (q,f) in γ, then γ can be written as $\gamma'(q,f)\gamma''$, ($\forall i$, $0<i<n$) $q_i \neq p$, ($\forall i$, $0<i\leq n$) $q_i \neq q$ and there is no arc (q_i,f) in γ', and there is no arc (q_0,f) in α. Thus, α, β, and γ' form a Z expression from p to q.

•

Case 4: If there is an arc progression α' from q_n to q which doesn't contain any arc (p,f) and there is no arc (q,f) in γ, then there is no arc (p,f) or (q,f) in γ, $p \neq q$, $(\forall i, 0 \leq i < n)$ $q_i \neq p$, and there is no arc (p,f) in α'. Thus, ϵ, γ, and $\beta\alpha'$ form a Z expression from p to q.

Case 5: If there is no such α' from q_n to q, then every arc progression from p to q which contains an arc (q_n,g) must also contain an arc (p,f). Therefore, there is also an arc progression γ' from p to q which doesn't contain any arc (q_n,g). As a result, $(\forall i, 0 < i < n)$ $c_i \neq p$, $(\forall i, 0 < i \leq n)$ $q_i \neq q$, $(\forall i, 0 \leq i < n)$ there is no arc (q_i,f) in α, and there is no arc (q_n,g) in γ or γ'. Thus, α, β, and $\gamma\gamma'$ form a Z expression from p to q.

Lemma 3.7.4: Assume the synchronization for a finite state resource M with initial state p has the property that there are arc progressions from some of the states to p. Let M' be a resource consisting of those states of M for which there is an arc progression to p along with all of the arcs which result in one of these states. Let M" be a resource consisting of the states of M not in M', a new state p', and the arcs of M not in M' with any state of M' replaced by p'. There are no states of M" in any arc of M'. M' is a persistent set with entry state p. M' and M" have disjoint sets of operations and are simple serial if M is simple serial. If the synchronization for M' can be expressed with the regular expression P', the synchronization for M" can be expressed with the regular expression R", and there is no Z expression from p to a final state, then there is no Z expression from p to p in M' or from p' to a final state in M" and the synchronization for M can be expressed by R'R".

Proof: By lemma 3.7.2 there can be no arc progression from any state of M" to any state of M' in M. Thus, every arc resulting in a state of M' must be of the form (q,f) for some state q of M'. If $(q_0,f_1)\ldots(q_{n-1},f_n)$ is an arc progression from a state q_0 to p in M, then $(\forall i, 0 \leq i < n)$ there is an arc progression from q_i to p. Therefore, q_i and (q_i,f_{i+1}) are in M' and there is an arc progression from q_0 to p in M'. Since there is an arc progression from every state of M' to p in M, there also is one in M' and M' is a persistent set. Since p is the initial state, it must be the entry state and also the only final state of M'. There is a one-to-one correspondence

between the arcs of M and those of M' and M". Therefore, if an operation is only part of one arc of M, it will be part of either one arc of M' or one arc of M". If there is a Z expression from p to p in M' there must also have been one in M and by lemma 3.7.3 there must have been a Z expression from p to a final state in M. Assume that α, γ, and β form a Z expression from p' to a final state in M". The first arc of α must be of the form (p',f) and results in a state q' of M". This arc must represent (q,f) in M where q is a state of M'. But then (q,f) is an arc progression from a state of M to a state of M" and by lemma 3.7.2 must contain an arc (p,g). Thus, q = p and α, γ, and β forms a Z expression from p to a final state in M. By theorem 3.1 and the fact that every arc (p',f) in M" represents an arc (p,f) in M, M can be represented by R'R".

Definition: The ***next set*** of a state p, N(p), is $\{q \mid (\exists f)\ S(p,f) = q\}$. The ***tail states*** of a state p is $\{q \mid (\forall q' \in N(p))$ there is an arc progression from q' to q$\}$. The ***tail arcs*** of a state p is $\{(q,f) \mid q$ is a tail state of p$\}$.

Lemma 3.7.5: If q is a tail state of some state p and there is an arc progression β from q to another state q', then q' is a tail state of p.

Proof: Since q is a tail state of p, for each state p' in N(p) there must be an arc progression α from p' to q. But then $\alpha\beta$ is an arc progression from p' to q'.

Lemma 3.7.6: If there is no Z expression from a state p to a final state, there are no arc progressions from p to itself, and there is at least one state in N(p), then there is a unique tail state p' of p such that every arc from p to a tail state of p other than p' must contain an arc (p',f).

Proof: Let α be an arc progression from p to a tail state p' of p which

contains no arc (q,f) for some tail state q. (If there is such a state q, let α be the arc progression from p to q instead). Assume that there is an arc progression from p to a tail state $q' \neq p'$ of p which contains no arc (p',f). Using the same argument as above, it may also be assumed that there is no arc (q,f) in the arc progression for some tail state q. If there is no such arc progression, then the proof is done. Either there is an arc progression β' from q' to some final state $q_f \neq p'$ which contains no arc (p',f) or else there is one from p' to a final state other than q' which contains no arc (q',f). Without loss of generality, it may be assumed that the former is the case. It should be noted that since there is no arc progression from p to itself, p isn't in N(p) nor can it be a tail state of itself. Since there is an arc progression from q' to q_f, by lemma 3.7.5 q_f must be a tail state of p. If there is an arc progression γ from q_f to p', then for each arc (q,f) in γ, by lemma 3.7.5, q must be a tail state of p and it must be true that $q \neq p$. Therefore, α, γ, and ϵ form a Z expression from p to q_f. Since there is no such Z expression, there can be no such γ. Let the last arc of α be (p",f). Since p" isn't a tail state of p there must be some state q" in N(p) such that there is no arc progression from q" to p". Let γ and β be the arc progressions from q" to p' and to q' respectively. For every state q such that (q,g) is an arc in α there trivially is an arc progression from q to p". Thus, there can be no arc (q,h) in γ or in β. Also, there can be no arc (q_f,h) in γ since otherwise there would be an arc progression from q_f to p'. It must therefore be true that α, γ, and $\beta\beta'$ form a Z expression from p to q_f.

Definition: The unique state p' will be called the ***tail entry state*** of p.

Lemma 3.7.7: Assume there is no Z expression from a state p to a final state, there are no arc progressions from p to itself, and there is at least one state in N(p). If p' is the tail entry state of p and q isn't a tail state of p but there is an arc progression α from p to q, then any arc progression β from q to a tail state of p must contain an arc (p',f).

Proof: Since $\alpha\beta$ is an arc progression from p to one of its tail states, by lemma 3.7.6 either α or β must contain an arc (p',f). If (p',f) is an arc of α, then there is an arc progression from p' to q and by lemma 3.7.5 q must be a tail state of p, a contradiction.

Lemma 3.7.8: Assume there is no Z expression from a state p to a final state, there are no arc progressions from p to itself, and there is at least one state in N(p). If p' is the tail entry state of p, then there can be no Z expression α, γ, and β from p to p' such that no arc (q,f) in α', γ, or β q is a tail arc.

Proof: There must be an arc progression β' from p' to a final state. By lemma 3.7.5, for each arc (q,f) in β' must be a tail arc of p. Thus, α, γ, and $\beta\beta'$ forms a Z expression from p to the final state.

Lemma 3.7.9: Assume there is no Z expression from a state p to a final state, there are no arc progressions from p to itself, and there is at least one state in N(p). If p' is the tail entry state of p, then there can be no Z expression α', γ, and β from p' to a final state such that every arc (q,f) in α', γ, or β is a tail arc.

Proof: By lemma 3.7.6, there must be an arc progression α from p to p' which contains no arc (q,f) for a tail state q of p. Thus, $\alpha\alpha'$, γ, and β would form a Z expression from p to the final state.

Lemma 3.7.10: If a finite state resource is such that there is no arc resulting in the initial state p, there is no Z expression from p to a final state, and there is at least one tail state of p, then every final state is a tail state of p.

APPENDIX

Proof: If there is an arc progression from a persistent set entry state q to the tail entry state p', then p' must be in the persistent set and there must be an arc progression from p' to q. Therefore, by lemma 3.7.5 q must be a tail state of p. Assume that there is a final state p" which isn't a tail state of p. Let $(p,f'')\beta$ be an arc progression from p to p" where β is from a state $q'' \in N(p)$ and contains no arc (p,g). Since p" isn't a tail state of p, $(\exists q' \in N(p))$ there is no arc progression from q' to p". Let (p,f') be an arc progression from p to q', α be one from q' to p', and γ be one from q" to p'. There can be no arc (p",g) in γ or α or else there would be an arc progression from p" to p'. If (q,f) is an arc in β, then there can be no arc (q,g) in α or else there would be an arc progression from q' to p". The arc progressions $(p,f')\alpha$, γ, and β therefore form a Z expression from p to p", a contradiction.

Lemma 3.7.11: Assume the synchronization for a finite state resource M with initial state p has the property that no arcs result in p and there is at least one tail arc of p. Let M' be a resource consisting of those states of M which aren't tail states of p, a new final state q', and every arc (q,f) for a state q of M' such that if an arc of M' results in a tail state of p in M then it results in q' in M'. Let M" consist of the tail states of M and all of the arcs (q,f) for a state q of M". The initial state p is in M'. If there is no Z expression from p to a final state in M, then every final state of M is in M", there is no Z expression from p', the tail entry state of p, to a final state in M", and if the synchronization for M' can be expressed with R' and the synchronization for M" can be expressed with R", then the synchronization for M can be expressed with R'R". If M is simple serial, then so are M' and M" and the sets of operations are disjoint.

Proof: Since no arc results in p, p isn't in N(p) and there can be no arc progression from a state in N(p) to p. Therefore, p isn't a tail state of itself and must be in M'. By lemmas 3.7.8, 3.7.9, and 3.7.10, if there is no

APPENDIX

Z expression from p to a final state in M, then every final state must be a tail state of p and therefore in M". Also, by lemmas 3.7.5 and 3.7.7, there is no arc progression from a state of M" to a state of M' in M and any arc progression from a state of M' to a state of M" other than p' in M must contain an arc (p',f). Using the construction from theorem 3.1 for an expression of the form RR' where there is only one state in F produces M from M' and M". Finally, there is a one-to-one correspondence between the arcs of M' and M" and the arcs of M. Therefore, if an operation is in only one arc of M, it will be in only one arc of either M' or M".

Definition: The *accessable arcs* of a state p, A(p), is {(q,f) | there is an arc progression from p to q}. Note that the tail arcs of a state p is the intersection over $q \in N(p)$ of A(q).

Lemma 3.7.12: Let p be a state such that no arcs result in p and there is no Z expression from p to a final state. For states $p_1,\ldots,p_n,p_{n+1} \in N(p)$ and $1 \le i \le n+1$, let $A_i = A(p_1) \cap \ldots \cap A(p_i)$ and $A_i' = A(p_i) \cap A(p_{n+1})$. If A_n isn't empty and ($\exists i$, $1 \le i \le n$) A_i' isn't empty, then either $A_n \subseteq A_i'$ and $A_{n+1} = A_n$ or else $A_i' \subset A_n$ and $A_{n+1} = A_i'$.

Proof: Assume that $(q,g) \in A_n$ and $(q',g') \in A_i'$ but (q,g) isn't in A_i' and (q',g') isn't in A_n. Since (q',g') isn't in A_n, ($\exists j$, $1 \le j \le n$) (q',g') isn't in $A(p_j)$. Let α be the arc progression from p_{n+1} to q', γ be the arc progression from p_i to q', and β be the arc progression from p_i to q. There also must be an arc progression from p_j to q but there can be none from p_{n+1} to q or from p_j to q'. Let β' be an arc progression from q to a final state q". Since no arc results in p, there can be no arc (p,f) in α, γ, or $\beta\beta'$. If q' = q" or if there is an arc (q",f) in γ, then there is an arc progression from p_j to q to q" to q' which is a contradiction. If there is an arc (p',f) in both α

and β or if (q',f) is in β then there is an arc progression from p_{n+1} to p' or q' to q which is a contradiction. Likewise, if there is an arc (p',f) in both α and β' or if (q',f) is in β' then there is an arc progression from p_j to q to p' or q' to q' which is a contradiction. Finally, since $p_{n+1} \in N(p)$, $(\exists f)\ S(p,f) = p_{n+1}$. The arc progressions $(p,f)\alpha$, γ, and $\beta\beta$' form a Z expression from p to a final state, which is a contradiction.

Lemma 3.7.13: Let p be a state such that no arcs result in p and there is no Z expression from p to a final state. For states q,q',q" $\in N(p)$, if $A(q) \cap A(q')$ isn't empty but $A(q) \cap A(q")$ is, then $A(q') \cap A(q")$ is empty.

Proof: Assume that $A(q') \cap A(q")$ isn't empty. Then by lemma 3.7.12, either $A(q) \cap A(q') \cap A(q") = A(q) \cap A(q')$ or else $A(q) \cap A(q') \cap A(q") = A(q') \neq A(q")$, neither of which is empty. (Let p_1=q, p_2=q', p_3=q", and i=n=2). Therefore, $A(q) \cap A(q")$ isn't empty, a contradiction.

Lemma 3.7.14: Let p be a state such that no arcs result in p and there is no Z expression from p to a final state. If $p_1,\ldots,p_n \in N(p)$ and A_i is as defined in lemma 3.7.12, then $(\exists j, 1<j\leq n)\ A_n = A(p_1) \cap A(p_j)$.

Proof: $A_2 = A(p_1) \cap A(p_2)$ so the lemma is true for n=2. Assume that for some n, n≥2, the lemma is true. Thus, $(\exists j, 1<j\leq n)\ A_n = A(p_1) \cap A(p_j)$. By lemma 3.7.12, if $A(p_1) \cap A(p_{n+1})$ isn't empty, then either $A_{n+1} = A_n = A(p_1) \cap A(p_j)$ or else $A_{n+1} = A(p_1) \cap A(p_{n+1})$. Likewise, if $A(p_1) \cap A(p_{n+1})$ is empty, then A_{n+1} is empty and $A_{n+1} = A(p_1) \cap A(p_{n+1})$. Thus, the lemma is also true for n+1 and by induction $(\forall n, 1<n)$ it is true.

Lemma 3.7.15: Assume the synchronization for a finite state resource M with initial state p has the property that no arcs result in p, there are no tail

arcs of p, and there is no Z expression from p to a final state. It may also be assumed that if for state q there is no arc (q,f) then there is only one arc (q',f) such that S(q',f) = q. If there is another arc (q",g) such that S(q",g) =q, then ereate a new state p" equivalent to q and let S(q",g) = p". Select any state p" such that p" ∈ N(p). Let B(p") = {q | q ∈ N(p) and A(p") ∩ A(q) isn't empty} ∪ {p"}. ({p"} is necessary in case A(p") is empty). Let M' be a resource consisting of p, B(p"), any state q' such that (∃q ∈ B(p")) there is an arc progression from q to q', any arc from p to a state q ∈ B(p"), and any arc (q',f) such that (∃q ∈ B(p")) (q',f) ∈ A(q). Let M" be a resource consisting ohf a new state p' and all of the states and arcs of M which aren't in M' with the exception that every arc (p,f) is replaced by (p',f). Either there is only one arc or else there must be at least one in M' and at least one in M". There is no Z expression from p to a final state in M' or from p' to a final state in M". If M is simple serial, then so are M' and M" and M' and M" are disjoint. Finally, if the synchronization for M' can be expressed by R' and the synchronization for M" can be expressed by R" then the synchronization for M can be expressed by R'R".

Proof: Since there are no tail arcs of p, the intersection over the states q ∈ N(p) of A(q) is empty. If p" is the only state in N(p), then there can be no arc (p",f). Otherwise the arc would be a tail arc of p. Since every arc (p,f) results in p" and there can only be one arc resulting in p", there is only one arc. Assume that there is at least two states in N(p). By lemma 3.7.14, (∃q ∈ N(p)) A(p") ∩ A(q) is empty. Thus, there are arcs (p,f) to p" in M' and (p,g) to q in M". By the definitions of M' and A(q), for each arc (q,f) in M' q is in M'. If q=p, then S(q,f) must be in B(p") and therefore also in M'. Otherwise, (∃q' ∈ B(p")) such that (q,f) ∈ A(q'). Thus there is an arc progression from q' to q to S(q,f) which therefore must also be in M'. For each arc (q,f) in M", either q=p' and S(q,f) is in N(p)-B(p") or else every arc progression from p to q in M starts with an arc (p,g) such that S(p,g) = q' and q' ∈ N(p)-B(p"). Thus, (q,f) ∈ A(q') and there is an arc progression from q' to S(q,f). Since A(p") ∩ A(q') is empty and

$(\forall q'' \in B(p''))$ $A(p'') \cap A(q'')$ isn't empty, by lemma 3.7.13, $A(q') \cap A(q'')$ is empty. Thus there can be no arc progression from q" to q. If there is an arc progression from q" to S(q,f), then any arc (S(q,f),g) would be in both A(q') and A(q"), a contradiction. Thus, either there is no arc progression from q" to S(q,f) or else there is no arc (S(q,f),g). But in the latter case, only one arc can result in S(q,f) and that is (q,f). Since there is no arc progression from q" to q, there can be none fron q" to S(q,f). Using the construction of theorem 3.1, the synchronization can be expressed as R'+R" and since each arc of M is either in M' or M", if M is simple serial then so are M' and M" and R' and R" must be disjoint. If for a state q of M' or M" there is no arc (q,f) then there can be no arc (q,f) in M. Likewise, every persistent set of M' and M" must be a persistent set in M with the same entry states. Thus, every final state of M' and M" must be a final state of M and every Z expression in M' or M" from p or p' respectively to a final state must be a Z expression from p to the same final state in M.

The proof of theorem 3.7 will now be given.

Proof: For a set of synchronization relationships M with initial state p there are five possibilities.

1. There is an arc progression from every state to p. In this case, the set of states form a persistent set with p as the only entry state. By lemma 3.7.1, the resource M' as described in that lemma must be simple serial with no Z expression from p to a final state. Also, if the synchronization for M' can be expressed with the regular expression R, the the synchronization for M can be expressed with the regular expression R^*. Finally, there are no persistent sets in M' and p isn't the successor of any arc in M' so R can't be either final nor initial loop.

2. There is an arc progression from some states to p. By lemma 3.7.4, the resources M' and M" as described must be simple serial with no Z expression from p to p in M' or from p' to a final state in M". If the synchronization for M' and M" can be expressed with R' and R" respectively, then the synchronization for M can be expressed with R'R". Finally M' is a persistent set so R' is simple final loop and p' isn't the successor of any arc in M" so R" isn't initial loop.

3. There is a single arc (p,f) to a state q. The synchronization can be expressed with the restricted regular expression f.

4. No arc results in state p and there is at least one tail arc of p. By lemma 3.7.11, the resources M' and M" as described must be simple serial with no Z expression from p to q' in M' or from p' to a final state in M". If the synchronization for M' and M" can be expressed with R' and R" respectively, then the synchronization for M can be expressed by R'R". Finally, the operations of R' and R" are disjoint and there is no arc (q',f) in M' so R' isn't final loop.

5. No arc results in state p and there are no tail arcs of p. By lemma 3.7.15, the resources M' and M" as described must be simple serial with no Z expression from p to a final state in M' or from p' to a final state of M". If the synchronization for M' and M" can be expressed with R' and R" respectively, then the synchronization for M can be expressed by R'+R". Finally, the operations of R' and R" are disjoint and p isn't the successor of any arc in M' and p' isn't the successor of any arc in M" so neither R' nor R" is initial loop.

This procedure of altering the synchronization relationships must eventually terminate since the total number of arcs is constant. Also, in cases 2, 4, and 5, the number of arcs in M' and M" must each be fewer than the number of arcs in M. Finally, after case 1 is used, one of the other cases applies to the result.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR - TR - 76 - 0853	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): SYNCHRONIZATION OF FINITE STATE SHARED RESOURCES	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Edward A. Schneider	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101D, AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, VA 22209	12. REPORT DATE: March 1976
	13. NUMBER OF PAGES: 130
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research (NM), Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
(12) 133 p.	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problem of synchronizing a set of operations defined on a shared resource is studied. It is assumed that the decision as to which operations may be executed at some given time is dependent only on the sequence in which the operations have already executed. Equivalence classes of these sequences, called states, can then be

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 403 081

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

PRECEDING PAGE BLANK-NOT FILMED

used to define synchronization. A restriction is made such that only those resources for which the synchronization can be expressed using a finite number of states will be studied. The states along with a successor function, which is defined for a state-operation pair if the operation may start execution when the resource is in that state, form what are called synchronization relationships.

A distinction is made between resources on which only one process may execute an operation at a time, called serial resources, and resources on which several processes may execute operations in parallel, called concurrent resources. To handle concurrent resources, the states must be modified so that they correspond to equivalence classes of sequences of perilogues instead of operations. A perilogue is either the start or the finish of the execution of some operation.

Several variations of regular expressions are presented with which the synchronization for a shared resource might be expressed. Also, a method which can be used to implement the synchronization relationships is given. This implementation has a high overhead so several possible simplifications are shown. Each variation of regular expressions and each simplification of the implementation is shown to correspond to some restricted class of the synchronization relationships. The set of synchronization problems which can be solved using one implementation or notation which can't be solved using some other implementation or notation can be found by comparing the corresponding classes.